Inductive Boolean Function Manipulation:

A Hardware Verification Methodology for Automatic Induction

Aarti Gupta

October 31, 1994
CMU-CS-94-208

Carnegie Mellon

Inductive Boolean Function Manipulation:

A Hardware Verification Methodology for Automatic Induction

Aarti Gupta

October 31, 1994
CMU-CS-94-208

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Thesis Committee:
Allan L. Fisher, Chair
Randal E. Bryant
Edmund C. Clarke
Robert K. Brayton, University of California at Berkeley

This research was sponsored by the Wright Laboratory, Aeronautical Systems Center, Air Force Material Command, USAF, and the Advanced Research Projects Agency (ARPA) under grant F33615-93-1-1330. The US Government is authorized to reproduce and distribute reprints for Government purposes, notwithstanding any copyright notation thereon.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of Wright Laboratory or the U.S. government.

Keywords: formal hardware verification, proofs by induction, automatic inductive verification, inductive Boolean functions, Binary Decision Diagrams, parametric hardware representation, sequential function manipulation, reverse finite state automata

Carnegie Mellon

School of Computer Science

DOCTORAL THESIS
in the field of
Computer Science

INDUCTIVE BOOLEAN FUNCTION MANIPULATION:
A Hardware Verification Methodology for Automatic Induction

AARTI GUPTA

Submitted in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy

ACCEPTED:

Allen L. Robinson — October 31, 1994
THESIS COMMITTEE CHAIR — DATE

[signature] — 1/19/95
DEPARTMENT HEAD — DATE

APPROVED:

R. Rashid — 1/20/95
DEAN — DATE

To my parents Kamla and Dhan Parkash

कर्मण्येवाधिकारस्ते मा फलेषु कदाचन
मा कर्मफलहेतुर्भूः मा ते संगोऽस्त्वकर्मणि

—भगवद् गीता, 2.47

Set thy heart upon thy work, but never on its reward.
Work not for a reward; but never cease to do thy work.

—Bhagavad Gita, 2 . 47

Acknowledgements

It is a daunting task to even attempt to enumerate the help and support of everyone to whom my thanks are due. Nevertheless, I shall try.

I am deeply grateful to Allan Fisher who has advised me well throughout the sometimes difficult and still on-going process of developing into a researcher. His constant support was especially critical during the initial stages, and it was on his suggestion that I first started started working on the problem of automatic induction. It is also a testimony to his versatile skills, that I received good guidance and feedback in all aspects of this work, ranging from the highly technical to the most practical. Undoubtedly, his insistence on the latter will continue to influence me in the future.

It was the fortunate occurrence of a course on formal verification offered by Randy Bryant, Ed Clarke and Allan Fisher at CMU that first attracted me to this area. It is no coincidence that my work is based mostly on their contributions. Given Randy's first-hand experience with symbolic BDD manipulation, his encouragement has been very valuable. No less valuable were his insightful suggestions during the many discussions we had. I am very grateful for his help and interest in my work. With Ed also, it has always been instructive to draw upon his considerable experience. His enthusiasm, both with students and with visitors, has contributed in no small measure to my own development. I am also thankful to him for partially sponsoring my conference travel on occasion.

I consider myself lucky for having the opportunity to interact with Bob Brayton from Berkeley. (His providing me with the distinction of having an advisor-in-law on the thesis committee is not the sole reason!) His extensive knowledge and familiarity with sequential logic design, coupled with his recent interest in formal verification, placed him ideally with respect to the span of my thesis research. His technical feedback and numerous suggestions, despite his busy schedule, helped to sharpen the focus of my work. I thank him and Aunt Ruth for their constant encouragement and support.

Other people contributed a great deal to the excellent environment I had at CMU. In particular, thanks go to David Long for generously allowing me the use of his BDD package, and also to Ken McMillan, Derek Beatty, Carl Seger, Somesh Jha for technical discussions on my preliminary work. Alok Jain and Rajen Ramchandani, who had little choice in this regard, also deserve my heartfelt gratitude! I also wish to acknowledge the wonderful presence of Sharon Burks, Catherine Copetas and Terri Stankus, that eased transitions through the departmental procedures for me. My thanks also to the facilities staff, and the entire community that makes SCS a wonderful place. I feel fortunate for having been a part of it.

Where would I be without friends? Its been wonderful to have friends that made school bearable even at its worst – Amy, Indira, Joao, Juan, Mark, Rick, and everyone at CTQC. Thanks also to those that helped me find a life beyond school – Anshu, Anu-Nattu, Anurag A., Anurag G.,

Chandra, Joyoni, Puneet, Rajiv-Sapna, Sangeeta, Shobana. More than friends, perhaps sisters from a yester-life – Udita and Siloni, thank you for always being there for me.

I would like to thank Santosh Aunty and Darshan Uncle for their constant encouragement. Manish, always a source of joy and pride, provided me more support than I ever thought younger brothers could. With Veenita accompanying him all the way – I thank them both for their love. For Sharad, it is difficult to find adequate words of gratitude. He provided me with the strength to get through it all. The PA Turnpike Commission got richer for our travails, but I felt richer still in having his love and support. I cannot thank him enough. Finally, and most importantly, I don't know where to begin to thank my parents. I could never have come this far along this path without their love, their understanding, and their sacrifice. I remain indebted to them forever.

Abstract

The high level of complexity of current hardware systems has led to an interest in formal methods for proving their correctness. This thesis presents a methodology for formal verification of inductively-defined hardware, based on automatic symbolic manipulation of classes of inductive Boolean functions (IBFs). It combines reasoning by induction and symbolic tautology-checking in a way that incorporates the advantages of both, where the key idea is that by building an inductive argument into the IBF representations, explicit proofs by induction can be avoided. The methodology consists of identifying classes of inductive Boolean functions that are useful for representation of hardware and specifications, associating a schema with each class (consisting of a canonical representation and symbolic manipulation algorithms), and appropriately manipulating these representations for performing formal verification.

The thesis research focusses on two classes of IBFs – linearly inductive functions (LIFs) and exponentially inductive functions (EIFs), intended to capture a serial style of induction and a divide-and-conquer style of induction, respectively. Their canonical representations and symbolic manipulation algorithms are based on extensions of Bryant's Binary Decision Diagrams (BDDs), where the complexity is independent of the induction parameters. The thesis also explores additional representation machinery required to capture structurally-inductive hardware in terms of LIFs and EIFs, including support for handling compositions of inductive descriptions.

The LIF schema is particularly useful since it naturally captures the temporal induction inherent in sequential system descriptions. It provides canonical representations and symbolic manipulations for sequential functions in much the same way as BDDs provide for combinational functions. An interesting aspect of the LIF representation is that it corresponds to a minimal reverse deterministic finite state automaton (DFA), i.e. a minimal DFA that accepts input strings in reverse order. Its practical usefulness is demonstrated by showing that in comparison to classic (forward) DFAs, several datapath circuits have exponentially more compact reverse DFAs.

Symbolic IBF manipulations can be used to perform both functional verification of size-parametric hardware (regular in structure), and behavioral verification of sequential circuits (regular in time). As an example of the former, a proof by induction for a specification property corresponds to performing a tautology-check on the specification IBF formula. For the latter, the LIF equivalence method can be directly used to check input/output equivalence of two differently-encoded finite state machines. Furthermore, the unified framework for handling induction in the structural and the temporal domains is useful for handling a combination of both, through techniques such as symbolic simulation of space-parametric circuits, and language containment for obtaining network invariants. The thesis includes a practical implementation of the outlined verification methodology, including a core package for symbolic LIF manipulation.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Technological advances in the areas of circuit design automation and fabrication have made available much larger hardware systems today, than ever before. As higher functionality circuits are designed, their complexity continues to grow, and it has long become impractical to reason about their correctness manually. Traditionally, simulation has been used to check for correct operation, whereby the observed response to a set of inputs is checked against the expected response, with any discrepancy indicating an incorrect design. However, using simulation as a verification method is now proving to be inadequate due to computational demands of the task involved. It is not practically feasible to simulate all possible input patterns to completely verify a hardware design. An alternative to post-design verification is the use of automated synthesis techniques supporting a correct-by-construction design style. Logic synthesis techniques have been fairly successful in automating the low-level (gate-level) logic design of hardware systems. However, more progress is needed to automate the design process at the higher levels in order to produce designs of the same quality as is achievable manually by circuit designers. Until such time as synthesis technology matures, high-level design of circuits will continue to be done manually, thus making post-design verification essential.

Typically, a much reduced subset of the exhaustive set of patterns is simulated after the design of a system. It is hoped that no bugs have been overlooked in this process. Unfortunately, this is not always the case in practice. Numerous instances exist of cases where errors have been discovered too late in the design cycle, sometimes even after the commercial production and marketing of a product. Such late detection of errors is a very costly proposition — not only in terms of lost time to market a product, but also in terms of a potential loss of production in case of product recall. These are compelling reasons for verifying hardware to be correct, and completely correct, right at the design stage.

A comparatively recent alternative to simulation has been the use of formal verification for determining hardware correctness. Formal verification is, in some sense, like a mathematical proof. Just as correctness of a mathematically proven theorem holds regardless of the particular

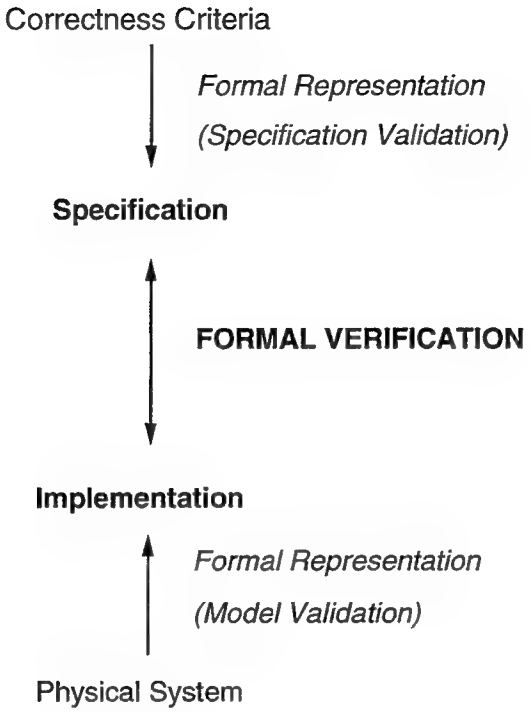


Figure 1.1: A Formal Verification Problem

values that it is applied to, correctness of a formally verified hardware design holds regardless of its input values. Thus, consideration of *all cases* is implicit in a methodology for formal verification. In addition to being theoretically sound, these methods have been demonstrated to work reasonably well in practice in a number of cases. Their success has attracted a fair deal of attention from both the research community and the industry, with exciting progress being made on many fronts.

1.1 What is a Formal Hardware Verification Problem?

Due to the overlap of different technical areas spanned by formal hardware verification – formal logic, language and automata theory, hardware design, design automation (to name a few) – there is some amount of variation in exactly what constitutes a formal verification problem. The view taken in this thesis is that a formal hardware verification problem consists of *formally establishing that an implementation satisfies a specification*, as shown pictorially in Figure 1.1. The term *implementation* (Imp), refers to a formal representation of the physical hardware system which we want to reason about. It can correspond to a hardware design description at any level of the hardware abstraction hierarchy, and not just the final layout (as is traditionally

regarded in some areas). Some of the typical representations used are networks of transistors or gates, finite state automata/machines, logic formulas etc. The exact choice usually depends on the abstractions used to model the physical hardware. The term *specification* ($Spec$), refers to a formal representation of the correctness criteria to be checked. It can also be represented in a variety of ways – as a behavioral description, as an abstract structural description, as a functional property, as a timing requirement etc. As for the *satisfaction relation*, there is considerable variation in its exact form too, which depends to a large extent on the formalisms used for representing the implementation and the specification. In some cases, for example, a strict equivalence is desired, i.e. $Imp \equiv Spec$. In others, a logical implication of the form $Imp \Rightarrow Spec$ suffices, whereby it is proved that every instance of the implementation satisfies the specification also. In yet others, the implementation may serve as a logic model in which the truth of the specification, expressed as a logic formula, holds, denoted $Imp \models Spec$. Various alternatives have been explored by numerous researchers, along each of the three dimensions consisting of – the implementation representation, the specification representation, and the satisfaction relation. Some of these are described in a survey article on formal hardware verification methods [67].

1.1.1 Specification Validation and Model Validation

Though a formal hardware verification methodology necessarily includes a choice of formal representations for the implementation and the specification, the focus within a given problem is usually on proving the required satisfaction relationship between the two. In particular, it does not directly address the problem of *specification validation*, i.e. whether the specification means what it is intended to mean, whether it really expresses the property one desires to verify, whether it completely characterizes correct operation etc. Sometimes, specification validation can be indirectly cast in terms of the formal verification problem itself as follows – a specification for a given verification problem can itself be made the object of scrutiny, by serving as an implementation for another verification problem at a conceptually higher level. The purpose of the latter problem is to test if the meaning of the original specification is as intended, the "intended" meaning thus serving as a specification at the higher level. Note that even this formulation implies a leap of faith at the highest level considered, where specifications are held to be infallible. However, this leap of faith is a necessary characteristic of any representation mechanism which attempts to bridge the gap between the formal and the informal.

Similarly, at the lowest end too, a formal verification methodology does not directly address the problem of *model validation*, i.e. whether the model used to represent the implementation is consistent, valid, correct etc. It is obvious that the quality of verification can only be as good as the quality of the models used. On the one hand, models should be faithful to the relevant characteristics of the physical system in order to ensure soundness of verification. Incorrect,

or partially correct, assumptions can frequently lead to misleading results – *false positives* (an incorrect design is wrongly labeled correct) or *false negatives* (a correct design is wrongly labeled incorrect). On the other hand, models should also abstract out the irrelevant details in order to manage complexity of the verification task. Simple models are generally more efficient than complex ones. The crucial requirement, and the tricky part, is to decide what is relevant and what is not in choosing the abstractions. Typically, a compromise between details and simplicity is necessary to make models practically useful.

Thus, the problem of formal verification is closely tied to the problems of specification validation and model validation. Typically, a formal verification methodology advocates a certain style of representation for the specification and the implementation, and the emphasis is on providing techniques for establishing the formal satisfaction relation between the two. Certainly, its scope of application and its practical success may critically depend on the features supported by the chosen representations. These are described in detail for the methodology advocated in this thesis. However, in general, the issues of representation are regarded as being outside the scope of a formal verification problem.

1.1.2 Relationship with Hardware Abstraction Hierarchy

An important feature of formal verification, as formulated in the previous section, is that it admits different verification tasks corresponding to successive levels of the hardware abstraction hierarchy. Typically, the design of a hardware system is organized at different levels of abstraction, the top-most level representing the most abstract view of the system and the bottom-most being the least abstract, usually consisting of actual layouts. Verification tasks can also be organized naturally at these same levels. An implementation description for a task at any given level, can serve as a statement of the specification for a task at the next lower level, as shown in Figure 1.2. In this manner, top-level specifications can be successively implemented and verified at each level, thus leading to implementation of an overall verified system. Hierarchical organization not only makes this verification process natural, it also makes the task tractable. Dealing with the complexity of a complete system description of even modest size these days, is out of bounds for most verification techniques. By breaking this large problem into smaller pieces that can be handled individually, the verification problem is made more manageable. However, it may not be possible for a single methodology to accomplish hierarchical verification across all levels. The currently available methodologies cover a subset of levels at best, and there are several efforts aimed at combining their strengths.

Some researchers have differentiated between verification tasks from an application viewpoint, corresponding to the two major design phases [5, 76]. In the first phase, an initial high-level design is described, typically using a hardware description language such as VHDL, or Verilog. This high-level design is verified by using *design verification* which answers questions of the form "Is what I specified what I wanted?". In the second phase, the initial description is

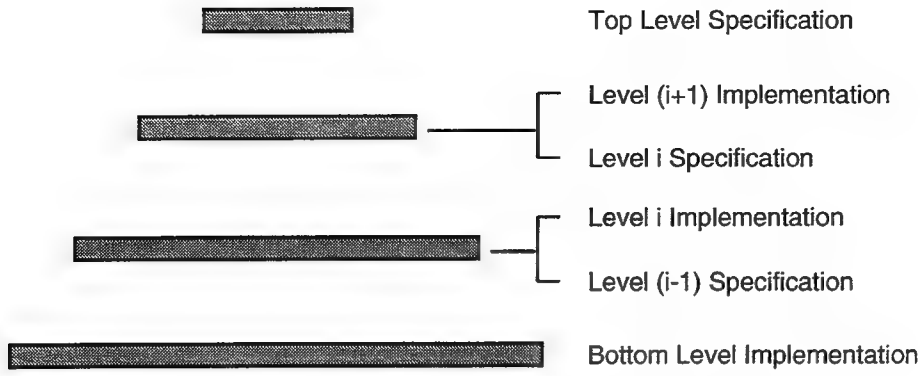


Figure 1.2: Hierarchical Verification Framework

synthesized into an implementation. *Implementation verification* is then used to answer the question "Is what I synthesized what I specified?".

While this view is suitable from the viewpoint of a complete design cycle, it reflects better on the relationship of a verification task to its location in the hardware abstraction hierarchy. At the higher levels, where the design details have still not been filled in, *design verification* tends to proceed by checking properties representing "what I wanted", sometimes also referred to as *property verification*. However, as remarked earlier, it is not possible to fully verify "what I wanted", and there is no choice but to *trust* the highest-level representations. At the lower levels of the hierarchy, verification tends to proceed in a different manner from property verification. Since most design systems use synthesis techniques, which have been shown effective in practice at medium and low levels, the verification requirement is to check correspondence between the different levels of the design that the synthesis techniques work with. For example, a typical verification task may involve checking the equivalence of a Register-Transfer-Level (RTL) description and a gate-level description. Thus, the verification tasks are typically different in the two phases, corresponding also to different levels in the abstraction hierarchy.

Both kinds of verification tasks can fit naturally within the formulation described earlier. In *design verification*, the property is regarded as the $Spec$, and the high level design is the Imp, which should satisfy the property. In *implementation verification*, the high level design description forms the $Spec$, and its equivalence with the lower level description, Imp, is checked. In either case, the required satisfaction relation is to be proved between $Spec$ and Imp, and the problems of specification validation and model validation at the two ends of the spectrum still remain.

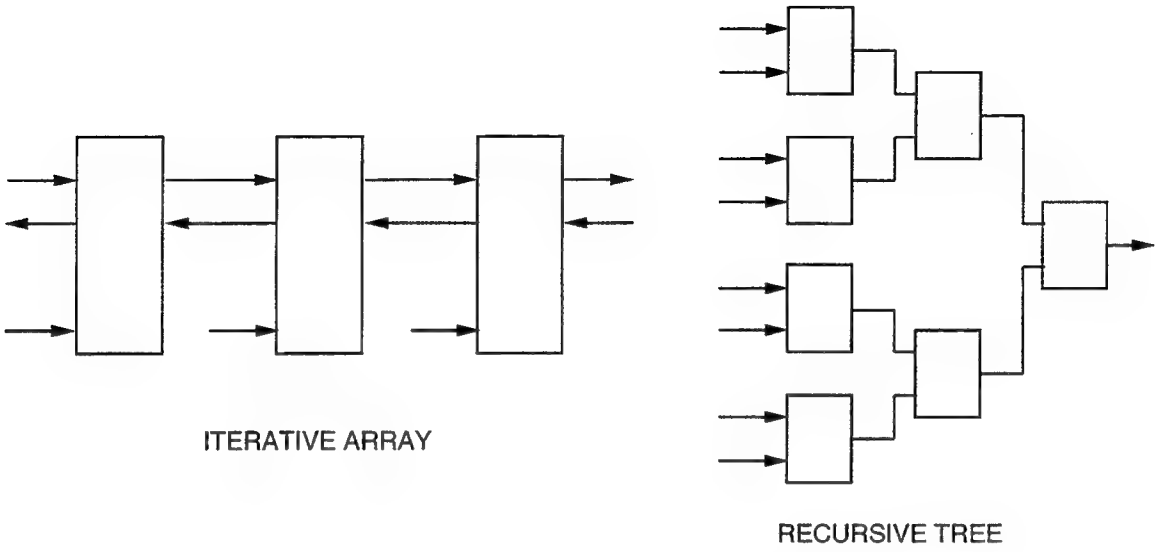


Figure 1.3: Inductively-defined Hardware Systems

1.2 Thesis Scope

The research work in this thesis has focused on developing a methodology for the formal verification of inductively-defined hardware systems. This methodology is based on automatic symbolic manipulation of *inductive Boolean functions (IBFs)*, and will be referred to as the IBF methodology from here on. Informally, an inductive Boolean function is a family of Boolean functions defined inductively. It can be used naturally to capture parametric descriptions of hardware, where the system structure (or behavior) can be described inductively in terms of formal parameters. The main feature of the IBF methodology is that it combines the strengths of reasoning by induction on one hand, and symbolic Boolean function manipulation on the other. The existing work in these areas, which provides the necessary background and motivation for the thesis, is reviewed in this section.

1.2.1 Inductively-defined Hardware Systems

The term "inductively-defined" refers to those systems where the hardware can be defined parametrically in terms of induction parameters. It includes iterative descriptions such as the linear array shown in Figure 1.3 on the left, as well as recursive descriptions such as the tree arrangement shown on the right.

Iterative hardware systems occur very naturally in practice. Most datapaths circuits can be described iteratively in terms of size parameters, where a fixed-sized instance of the circuit

can be obtained by serially putting together bit-slices of the basic circuit structure. Similarly, at a higher level of abstraction, the interaction among an ensemble of finite state machines may be described by means of an iterative control structure. For example, a daisy-chain priority scheme among multiple processors which use a common resource, can be described iteratively in terms of the number of processors. The potential advantages of structural regularity in iterative systems has attracted numerous studies on the analysis and synthesis of such systems [74, 91, 145].

In addition to parameterization in the structural domain, iterative circuits can also capture parameterization in the temporal domain. In fact, all finite state sequential systems can be regarded as iterative systems in the sense that they compute functions iteratively over time. The correspondence of structural iterative systems and finite state sequential systems was first observed by Huffman [82], and related to standard finite state machine models by McCluskey [107]. Thus, verification techniques for structural iterative systems extend directly to finite state sequential systems, which are of prime importance in the area of digital systems.

The class of recursive hardware systems naturally captures a divide-and-conquer style of function computation. The most popular form is a binary tree arrangement of basic circuit modules. In comparison to iterative systems, which perform serial computation, recursive systems offer the advantages of parallel computation. This has been the prime motivation for the study of popular recursive circuits (e.g. a tree carry-lookahead adder), which are faster than the functionally equivalent iterative circuits (e.g. a ripple carry adder). General techniques for obtaining equivalent recursive tree realizations from iterative circuit descriptions have been studied [145]. Such techniques are also used in correctness-preserving transformations, an essential part of most techniques for automated logic synthesis [149, 150]. Thus, formal techniques for recursive systems not only serve an interesting class of practical systems, but also provide a framework for examining its relationship with iterative systems.

From a practical standpoint, use of parametric designs in the form of standard libraries has recently been emphasized as an emerging trend [50]. Reuse of existing designs has become an important issue in practice, in order to meet demands of a quick turnaround time. Though a complete system may not have an obvious inductive description, large parts of it can be frequently obtained as instances of standard parametric designs. For example, n-bit array multipliers, n-bit comparators, n-bit ALUs are typically used in most hardware designs. Furthermore, in applications such as digital signal processing, all values of n may be needed, and not just $n = 4, 8, 16, 32, 64$ etc. By directly addressing the issue of parameterization in hardware system descriptions, the IBF methodology provides the necessary framework for representation and verification of such designs.

1.2.2 Formal Verification using Reasoning by Induction

All techniques used for formal verification of inductively-defined hardware systems use some form of reasoning by induction. In general, reasoning by induction is a very powerful proof technique, with the main advantage that a single proof suffices to prove the correctness of an entire family of designs. Furthermore, since the complexity of this proof is independent of the value of the induction parameter, it can be useful for reasoning about a single design instance also, in cases where this instance may be too difficult to handle in practice.

Among various formal verification techniques that use reasoning by induction, the differences arise mainly due to the context within which this reasoning takes place. Broadly speaking, there are two main categories:

- Automatic theorem-proving with formal logic:
  The specification and the implementation are represented as formulas in a formal logic, and the required relationship (logical equivalence, logical implication) is established by proving a theorem within a proof-theoretic framework. The proof is obtained by using well-defined rules of inference, and typically draws upon axioms which capture the relevant hardware abstractions. The currently popular theorem-provers for hardware verification include the Boyer-Moore theorem-prover [17], the HOL proof system [64], PVS [121].

- Model checking and Language containment:
  In model checking, the specification is represented as a formula in formal logic. For example, temporal logic is very useful for expressing several classes of correctness properties including safety properties, liveness properties, precedence properties [53, 106]. Typically, the implementation is represented as a state-transition description of hardware. The semantics of the logic is used to check that the implementation provides a *model* for the formula, i.e. the logical truth of the specification formula holds in the model provided by the implementation [41, 98, 127].

  Language containment techniques also use a state-transition description for the implementation. The specification can be represented either as another state-transition description or directly as a language. For example, ω-regular languages [141] can conveniently express several classes of correctness properties. Correctness, in this case, is viewed as a language containment relation between the languages of the implementation and the specification, i.e. $\mathcal{L}(Imp) \subset \mathcal{L}(Spec)$ [72, 77, 95].

  Both model checking and language containment techniques are based on (either explicit or implicit) state-space traversal of the implementation representation. They are strongly related due to the underlying automata theory common to both [76, 147, 156].

1.2.2.1 Automatic Theorem-proving vs. Model Checking

Theorem-proving is a deductive process, and there are both theoretical and practical concerns regarding management of its complexity. Automation can be, and has been, provided to some extent, e.g. in the form of rewrite rules, specialized tautology-checkers etc. However, the "automated" theorem-provers available today are semi-automated at best, in that they require some form of human interaction [17, 64, 121]. In effect, theorem-"provers" are more like theorem-"checkers" in most cases, and require great skill on the part of the user to guide the proof searching process. On the other hand, theorem-proving systems are very general in their applications. Formal logic allows representation of virtually all mathematical knowledge in the form of domain-specific theories, and reasoning about them using a common set of inference rules provides a unifying framework within which various verification tasks can be performed.

Model-checking, in contrast, is a relatively modest activity. Since attention is focussed on a single model, and there is no need to encode incidental knowledge of the whole world, the complexity of this task is much more manageable. In most cases, a concise algorithm can be provided which can be made completely automatic [41, 98, 127]. These algorithms usually have a counter-example mechanism also, i.e. if the implementation fails to satisfy the specification, an input trace demonstrating the actual failure can be obtained. This feature greatly facilitates diagnosis of errors and subsequent design rectification, and is very important from a practical standpoint. The drawback of such techniques is that they are not general in the way that theorem-provers are. Some efforts have been made towards unification of model-checking ideas in terms of automata theory, but the techniques largely remain domain-specific. Furthermore, these techniques do not scale well due to their reliance on a state-based representation. An increase in the number of system components leads to the problem of *state explosion*, limiting their use for practical designs.

Apart from the differences of inherent complexity and degree of automation between these two categories of techniques, there is also an issue of practical applicability. Since formal logic theorem-proving techniques typically use a structural representation of hardware, they are suitable for reasoning about functional specifications of parameterized hardware. On the other hand, model-checking approaches typically use a state-based hardware description which is oriented towards expressing its behavior (i.e. what atomic propositions are true in each state), rather than its structure. These models allow relatively easier formalization of behavioral issues such as concurrency, fairness, communication and synchronization. Thus, theorem-proving approaches have traditionally performed better at verification of functional specifications of datapaths, while model-checking approaches have been better at reasoning about the control aspects of a system.

The existing techniques for reasoning by induction in these two contexts are described in some detail in the following subsections.

1.2.2.2 Reasoning by Induction with Automatic Theorem-Provers

The most successful applications of induction with automatic theorem-provers have been seen in association with the Boyer-Moore Logic and its theorem-prover [17]. Boyer-Moore Logic is a quantifier-free[1] first-order logic with equality. Apart from the primitive logical constants (*true, false*), a user is allowed to introduce inductively-constructed object types, characterized by a set of axioms using the *Shell Principle*. According to this principle, all such type definitions should be accompanied by functions to recognize, construct and access different fields of the object type. A user is also allowed to introduce axioms that define these functions. In order to avoid inconsistencies, these axioms should satisfy the *Principle of Definition*. This principle ensures that all new functions are defined either non-recursively in terms of pre-defined functions, or in the case of recursive definitions, a well-founded ordering exists on some measure of the arguments that decreases with each recursive call. Standard inference rules for propositional logic [59] are used for reasoning in this logic. In addition, there is an inference rule for using the principle of induction, which relies on the same well-founded ordering as is used by the definition axioms. It is the incorporation of this rule that allows Boyer-Moore logic to reason about inductively-defined systems.

The Boyer-Moore theorem-prover provides an automated facility for generating proofs in the described logic. Successful applications in the area of hardware verification include verification of parameterized hardware designs by German and Wang [63], verification of FM8501 (a micro-programmed 16-bit microprocessor) by Hunt [83, 84], and verification of parameterized hardware modules within an automated synthesis environment by Verkest and others [149, 150]. All these efforts have made good use of the recursion and induction principles allowed by Boyer-Moore Logic, to reason about hardware functions with arbitrary-sized arguments. However, the process of proof generation, in particular the process of generating a proof by induction, is not fully automatic. Typically, the theorem-prover needs assistance from the user for setting up intermediate lemmas and helpful definitions. Another limitation is that low-level circuit aspects, such as charge-sharing, bidirectionality etc., are generally difficult to capture directly with logic predicates that represent circuit elements. To effectively combine both high- and low-level circuit effects, either better circuit models within Boyer-Moore Logic are needed, or an interface with other appropriate systems is necessary. One such hybrid system called VOSS [90], which includes an interface with a switch-level symbolic simulator called COSMOS [28], has been explored by Seger and Joyce. However, the process of proof generation still remains semi-automated at best.

Another popular automatic theorem-prover is the HOL system, based on a higher-order logic called HOL, developed by Gordon for the purpose of hardware specification and verification [37, 64]. The HOL logic is embedded in an interactive functional programming language called ML [65], which has expressions that evaluate to terms, types, formulas and theorems of HOL's

[1] There is implicit universal quantification of formulas with free variables.

deductive apparatus. The HOL system supports a natural deduction style of proof [59], with derived rules formed from eight primitive inference rules. In addition, there are special rules for help in automatic theorem-proving, e.g. a collection of rewrite rules. Most proofs done in the HOL system are goal-directed, and are generated by assistance from the user. Unlike the special inference rule in Boyer-Moore Logic, the principle of induction can be simply stated in HOL by the following higher-order formula:

$$\forall P.\ ((P\ 0) \wedge (\forall n.\ (P\ n) \Rightarrow (P\ (n+1)))) \Rightarrow \forall n.\ (P\ n)$$

The above formula asserts that for all properties P, if P is true for 0, and if P being true for n implies that it is also true for $(n+1)$, then P is true for all n. It is not possible to express this statement in first-order logic, since quantification over predicates (property P in this example) is not allowed.

In the HOL approach to hardware representation, circuits can be described in terms of both their behavior and structure [37]. The behavior of hardware devices is described by higher-order predicates, which hold true for a set of arguments exactly when these argument values could occur on the corresponding ports of the device. Their structure is described by using logical connectives on primitive components, which are typically represented at the switch and gate levels. Description of sequential circuits involves reasoning explicitly about time, and the axiomatization of primitive recursion allows representation of parameterized hardware. Other packages have also been formulated for defining recursive data-types, and for reasoning about them using induction [112]. This allows verification of parameterized hardware e.g. an n-bit adder, tree-structured circuits etc. However, higher-order logic systems suffer from some disadvantages too. The increased expressiveness carries with it a price tag of increased complexity of analysis. One disadvantage is the incompleteness of a sound proof system for most higher-order logics[2], e.g. incompleteness of standard second-order predicate logic [59]. Also, inconsistencies can easily arise in higher-order systems if the semantics are not carefully defined [73]. This makes logical reasoning more difficult than in the first-order case, and one has to rely on ingenious inference rules and heuristics, typically under guidance from the user.

1.2.2.3 Reasoning by Induction with Model Checking/Language Containment

As mentioned earlier, both model checking and language containment techniques use a state-transition graph representation of the hardware system to be verified. One of the serious limitations of these techniques is the reliance on an *explicit* state-transition graph. Typically, the number of states in such a graph increases exponentially with the number of gates/processes/elements/components in the system. This results in what is popularly called the *state explosion problem*, thereby limiting the application of these techniques to small circuits only. Interest in using induction grew in an attempt to alleviate this problem, at least in the

[2] with respect to standard models [59]

case of a system with an arbitrary number of identical processes. Some of these techniques are described in the remainder of this section. Another alternative is the use of symbolic methods, which use an *implicit* state-space representation, thereby allowing a non-enumerative approach for state-space traversal. These methods are described in detail in the next section.

Apt and Kozen proved that it is not possible, in general, to extend verification methods for a finite state process, in order to reason about an arbitrary number of processes [3]. Though this result sets the limit for using induction in the general case, it has prompted researchers to address special cases of the general induction problem.

Clarke, Emerson and Sistla demonstrated the effectiveness of model checking with Computation Tree Logic (CTL), a branching time temporal logic [40, 41]. Subsequently, several extensions such as CTL* were studied [53], with varying degrees of expressiveness and complexity. In an attempt towards incorporating induction, Clarke, Grumberg and Browne introduced a variant of CTL*, called Indexed CTL* (ICTL*). This logic allows formulas to be subscripted by the index of the process referred to [44]. A notion of *bisimulation* is used to establish correspondence between the models (Kripke structures) of two network systems with different number of processes, such that an ICTL* formula is true in one if and only if it is true in the other. However, the bisimulation relation needs to be provided by the user, and the state explosion problem is also not avoided (since the bisimulation relation itself uses the explicit state-transition relations). The notion of correspondence between Kripke structures was later extended, such that a "process closure" captures the behavior of an arbitrary number of identical processes [43]. Reasoning with the process closure allows establishment of ICTL* equivalence of all systems with more than a finite number of processes. However, again, this process closure has to be provided by the user. The network topologies handled by these techniques were also extended to classes expressible by context-free network grammars, with some further restrictions on the temporal logic [135].

Sistla and German also addressed this problem [137] in the context of concurrent CCS processes [113]. They have given fully automatic procedures to check if all executions of a process satisfy a temporal specification (given in a propositional linear time temporal logic [53]) for two system models – one consisting of an arbitrary number of identical processes, and the other consisting of a controller process and an arbitrary number of user processes. These algorithms can also be used for reasoning about global properties, e.g. mutual exclusion, and about networks of processes, e.g. token rings. However, the high complexity of these algorithms (polynomial and doubly exponential in process size, respectively) limits their practical application.

Within the language containment paradigm, McMillan and Kurshan incorporated reasoning by induction into an existing framework for checking ω-regular properties called COSPAN [96]. Essentially, an "invariant" process is used to perform the inductive step along a partial order provided by the language containment relation. This method applies in general to other process algebras, as well as other partial order relations. However, a limitation is that an invariant may be hard to find, since it has to represent the joint behavior of an arbitrary number of processes.

A similar method, using network invariants in the context of a general process theory, was independently proposed by Wolper and Lovinfosse [155]. McMillan and Kurshan's technique was later extended to also handle some cases where a process may depend upon a parametric number of other processes [97]. Some recent work in this area focusses on automatic generation of network invariants [6, 131], and is described in more detail in Chapter 6.

A slightly different problem from induction, but also addressing the issue of reasoning about an infinite number of data values, was explored by Wolper [154]. He showed that a large class of process properties stated over an infinite number of data values, are equivalent to those stated over a small finite set, provided the process is *data-independent*. Informally, a process is data-independent if its behavior does not depend upon the value of the data. (In general, determining data-independence for a process is undecidable, but certain syntactic checks can be used as sufficient conditions.) This has been used to specify correctness of a data-independent buffer process, i.e. given an infinite sequence of distinct messages it should output the same sequence, by showing that it is enough to specify the buffer for only three distinct messages. (An unbounded buffer cannot be characterized in propositional temporal logic otherwise [136].) This significantly adds to the specification power of propositional temporal logic, and also extends the applicability of various verification methods.

Another related method proposed by Clarke, Grumberg and Long is also based on the use of data abstractions, along with model checking of formulas in $\forall$CTL*, another variant of CTL* [45]. Data abstractions constitute a homomorphism from a given model of a system (with a large number of data values) to an abstract model (with a small number of data values), such that the truth of a $\forall$CTL* formula in the abstract model implies its truth in the original model. In practice, a conservative approximation of the abstract model is obtained by automatic symbolic execution of a high-level program over the abstract domain (by using abstract interpretations of the primitive relations). This method is particularly useful for reducing complexity of verification of datapaths, as has been demonstrated by its application to multipliers, a pipelined ALU etc.

1.2.3 Formal Verification Using Symbolic Boolean Function Manipulation

Symbolic methods for Boolean function manipulation have lately attracted a great deal of interest amongst researchers in all areas of digital design such as synthesis, simulation, testing and verification [27]. Their success is due largely to availability of a canonical Boolean function representation, such as Bryant's Binary Decision Diagram (BDD), and symbolic manipulation algorithms that are efficient in practice [12, 22]. Within the area of formal verification, the use of symbolic methods is almost orthogonal to other verification issues, in the sense that it allows compact internal data representations without affecting the high-level verification algorithms. This potentially results in an increase in efficiency and practical usefulness.

In its most direct form, symbolic Boolean function manipulation has been used for functional verification of combinational circuits [22, 23, 25]. Circuit outputs are represented as symbolic Boolean formulas on Boolean-valued symbolic variables that represent circuit inputs. Using the canonicity property of the Boolean function representations, it is trivial to check equivalence between an implementation and a specification. Symbolic methods have also been used for Hoare-style verification of finite state sequential circuits [8, 15, 24]. The states of a given circuit (implementation) are encoded by symbolic Boolean variables. The circuit's sequential behavior is described in the form of next-state functions and output functions over state variables and other variables representing the circuit inputs and outputs. Verification is performed by checking post-conditions of the Hoare-style assertions (specifications), for all transitions corresponding to states and inputs that satisfy the pre-conditions. The use of a symbolic simulator like COSMOS [28] allows multiple transitions to be simulated in one step. It also avoids construction of an explicit state-transition graph for the given circuit.

Extending the ideas from single transitions of a sequential system to multiple transition sequences, these methods have been used to perform automatic state-space exploration also. Again, the states are encoded by symbolic Boolean variables, and the transition behavior is described in terms of a next-state function (or a transition relation). The key idea is that a Boolean formula over state variables implicitly represents a set of states, i.e. those states that correspond to valuations of variables that make the formula true. This allows handling of *sets of states*, rather than individual states. For example, set intersection is performed by conjunction, and set union by disjunction, of the corresponding Boolean formulas. Finding the set of successor states to a given set of states, is handled easily by composition with the next-state function (or by a relational product using the transition relation). Finding the set of all reachable states is accomplished by iteratively finding the set of successor states, till a fixed point is reached. Again, the canonicity property of the representations enables simple syntactic checks to determine Boolean functional equivalence, and also allows efficient graph algorithms for symbolic manipulations in practice.

The main advantage of symbolic, as opposed to enumerative, state-space exploration is that it avoids construction of an *explicit* state-transition graph. The implicit representations allow the regularity in state-space of some circuits, e.g. datapaths, to be captured succinctly. This facilitates verification of much larger circuits in practice than is possible with direct state enumeration, and is especially useful in tackling the problem of state explosion encountered in systems consisting of a number of parallel components. It has been applied within the framework of several verification efforts. At one end of the spectrum, the representation of the next-state function has been completely avoided by using a symbolic simulator (COSMOS), which allows verification of finite symbolic trajectories [7, 30]. On the other, by using implicit state-transition functions (or relations), finite state machine equivalence can be verified by symbolic reachability analysis of the product machine [47, 48, 55, 94, 123, 124, 142]. Another application is in the area of model checking for temporal logics such as CTL. Each state

corresponds to a unique valuation of the atomic propositions, represented by symbolic state variables. By exploiting the fixpoint interpretations of temporal modalities, the set of states that satisfy a given temporal formula can be computed symbolically as a fixpoint of a Boolean functional on state variables [16, 32, 49, 108]. This method has been effective in discovering critical design errors in some real-life industrial applications, as demonstrated by verification of the cache consistency protocols for the Encore Gigamax [110], and the IEEE Futurebus+ standard [38]. A generalization to fixpoint operators allows symbolic model-checking of Mu-Calculus formulas [31], which provides a unified framework for model checking of various logics. Symbolic Boolean manipulation has also been used to implement verification techniques based on language containment [76, 77, 95].

Though symbolic methods can handle bigger problems than explicit enumeration methods, there is still a large gap with respect to handling realistic designs. Therefore, subsequent research in this area has focussed on various ways to handle bigger problems within the framework of symbolic Boolean function manipulation. Some are general techniques that have been known to work well otherwise, and others are specifically oriented to symbolic BDD manipulation:

- compositional and modular reasoning [66, 104]
  (based on an "assume-guarantee" paradigm [89, 125])
- use of abstraction mappings [36, 45, 95, 104]
  (based on the general concept of data abstraction [75])
- exploitation of symmetry [42, 54, 86]
- effective BDD techniques – e.g. partitioned transition relations [35], early variable elimination [34, 142], implicitly-conjoined invariants [80], dynamic variable reordering [132], breadth-first BDD manipulation [4, 119]
- symbolic verification systems – e.g. SMV (Symbolic Model Verifier) [108, 109], Murϕ [81, 52], HSIS (Hierarchical Sequential Interactive System) [5]

1.3 Thesis Motivation

The major motivation for the research reported in this thesis was to combine reasoning by induction and symbolic Boolean function manipulation. As described in the previous section, existing techniques that use reasoning by induction are semi-automated at best. They typically require assistance from the user, either in terms of guiding an automatic theorem-prover, or in terms of providing induction invariants. The use of automatic theorem-provers presents several problems in practice. First, their inherent computational complexity is very difficult to manage, and the generality of reasoning makes them quite inefficient even for easy tasks.

Second, the lack of complete automation places a great burden on the user, both in requiring familiarization with the underlying theory and inner workings of the proof system, and in learning the heuristics and practical tricks that might work where direct proof methods fail. This makes the task of using automatic theorem-provers daunting even for simple proofs. The use of induction invariants is more straightforward. Also, fully automated methods are usually made available for checking invariants and for conducting proofs with them. However, the burden of *finding* invariants is still placed largely on the user, who has to rely mostly on intuitive knowledge of the design and details of the proof technique.

Symbolic Boolean function manipulation, on the other hand, is fully automatic. Though the problems of representing and manipulating canonical Boolean functions are of NP-hard complexity [60], there are algorithms that have been found efficient in practice [22, 27]. However, most existing formal verification techniques based on symbolic Boolean function manipulation (as described in the previous section) deal only with fixed-sized circuits. Very few attempts have been made to extend their applicability to handle parametric circuits, e.g. those defined parametrically in terms of size parameters. The attempts that have been made [130, 131][3], have focused only on efficient representations for the functions in an iterative system. General symbolic manipulation has not been emphasized at all. Furthermore, the results have not been extended to other classes of inductive systems, such as recursive systems. Though the focus is on formal verification using induction, no attempt has been made to combine reasoning by induction with symbolic Boolean function manipulation at the operational level, as is done in the IBF methodology.

Thus, the potential advantages of combining reasoning by induction with symbolic Boolean function manipulation are – the ability to verify all instances (sizes) of a parametric circuit on the one hand, and complete automation and practical efficiency on the other. Furthermore, in existing techniques it is frequently the case that induction has been successfully used to prove the correctness of individual hardware units, while no mechanism is provided to reason similarly about a *composition* of inductively-defined units. One of the goals for the IBF methodology was to address this problem also, where the approach was guided by providing a general framework for symbolic manipulation (and not just verification) of inductively-defined systems. Finally, with the extensive efforts already made in symbolic verification of sequential systems, and with the natural relationship of sequential systems to iterative systems, it is interesting to explore general techniques that apply to both the structural and the temporal domains. Note that this would also allow a unified framework for induction in the structural domain (traditionally performed by theorem-provers), induction in the temporal domain (performed by behavioral verification techniques such as symbolic backward traversal), and a combination of both (such as language containment techniques with network invariants).

[3]These are also described in detail later, along with details of the IBF methodology, in order to clearly bring out the differences.

1.4 Thesis Overview

The basic approach used in the IBF methodology for combining induction with symbolic Boolean function manipulation, is to start from the latter end of the spectrum. Just as BDDs can capture arbitrary Boolean functions in a canonical form, the approach uses *inductive* representations that can capture *inductive* Boolean functions in a canonical form.

Different *classes* of IBFs are defined, based on the exact form of the parametric definition. The focus is on those classes that are useful for representation of inductively-defined hardware and specifications in practice. Each class of IBFs is associated with a *schema*, which consists of a canonical representation and symbolic manipulation algorithms for IBFs in that class. These schemas can be regarded as inductive extensions of BDDs. *The key idea is that by building the principle of induction into the IBF schema itself, an explicit proof by induction can be avoided.* In other words, a proof by induction is implicitly performed by symbolic manipulation of inductive Boolean functions, thereby freeing the user from the task of explicitly performing the proof.

Note that this approach does *not* claim to address the general problem of induction in a first-order (or higher-order) logic setting, as some theorem-proving systems with formal logics have been geared towards [17, 37, 85]. Only certain classes of inductive Boolean functions are considered, thereby avoiding the price of full generality. Nonetheless, the practical examples described in the thesis demonstrate that these classes can capture a wide variety of circuits and a range of verification tasks, making them useful in practice. This approach is also extensible in the sense that new classes of IBFs can be added, as and when necessary. Additional technical machinery may be required to handle them, though the main ideas and algorithms described in the thesis can be generalized to a large extent.

Thus, symbolic manipulation of IBFs is at the heart of the IBF methodology for formal verification of inductively-defined hardware systems. The main components of this methodology can be naturally viewed as:

- Characterization of useful classes of IBFs and the associated schemata
- Representation of inductively-defined hardware systems in the form of IBFs
- Applications of symbolic IBF manipulation to formal verification tasks

Each of these components is briefly described in the following sections, with pointers to parts of the thesis which describe them in detail.

1.4.1 Characterization of Inductive Boolean Functions

Two classes of IBFs are considered in this thesis – linearly inductive functions (LIFs) and exponentially inductive functions (EIFs). Informally, LIFs capture a linear (serial, iterative) style of induction, and EIFs represent a divide-and-conquer (parallel, recursive) style. Formal definitions and examples for these classes are given in Chapter 2.

The schemata for these IBF classes, which include canonical representations and symbolic manipulation algorithms, are described in Chapter 3. These are based on extensions of BDDs. The important feature of these representations and algorithms is that their complexity is independent of the parameter of inductive description.

The class of LIFs can naturally capture parametric circuits where the each circuit output is described iteratively in the structural domain, in terms of size parameters, e.g. ripple carry adder, barrel shifter, comparator etc. In this case, the LIF representation provides a constant-size representation for all size instances of the circuit. LIFs can naturally capture sequential circuits also, where the behavior of each sequential function is described iteratively in the temporal domain. In particular, the next-state and output functions of a deterministic finite state machine (FSM) can be regarded as LIFs, where time serves implicitly as the induction parameter. In this case, *the LIF representation directly provides a canonical representation for a sequential function, in much the same way as a BDD provides a canonical representation for a combinational function*. The canonicity property for FSM outputs is particularly useful, since it relates to the underlying sequences of FSM inputs, and does not rely on the number of states or a particular state encoding. Furthermore, the LIF symbolic manipulation algorithms provide a framework for handling regular languages (sequential functions) in a manner similar to combinational functions – language intersection/union/complementation are captured as simple graph algorithms for the Boolean-AND/OR/NOT operations on the LIF representations. Composition of FSMs is easily captured by LIF composition. Though these operations are similar to those for state transition graphs, there are significant differences between the two representations (described in detail in Chapter 5). The general symbolic manipulation framework for sequential functions can potentially find applications in areas other than formal verification, such as in synthesis, testing and simulation of sequential systems.

The class of EIFs is suitable for capturing recursive systems, such as a tree parity circuit, a log shifter, the carry-propagate and carry-generate functions of a tree carry-lookahead adder etc. In order to obtain canonicity in the EIF representations, a further restriction is placed on the EIF definition. This restriction allows the transfer of only a single bit of information from the left half of the circuit to the right half for computing the circuit outputs. Though this may seem restrictive, many practical recursive circuits fall within this class. This restriction can also be relaxed within a more general representation framework for IBFs. Both the LIF and EIF representations form special cases of this general framework, thereby indicating potential for a unified schema for different classes of IBFs.

1.4.2 Representation of Inductively-defined Hardware Systems

Some practical inductively-defined hardware systems can be directly captured in the form of LIFs and EIFs. However, additional representation machinery is required to handle general parameterization issues such as multiple parameters, parametric inputs, multiple outputs, vectors of outputs etc. Special attention is also needed in handling interaction of parameters in compositional descriptions. These additional mechanisms are described in detail in Chapter 4.

An important extension of the single parameter IBF schema is in handling multiple parameter IBFs. This is accomplished by using a general geometric hyperspace framework to capture induction trajectories employed in the definition of a multiple parameter function. The emphasis, again, is on enforcing practical restrictions that allow canonical representations, e.g. use of orthogonal trajectories only.

In the case of parametric circuits with multiple outputs, the goal of using a constant-size representation for all size instances may require use of a parametric output index. In some cases, this output index can be regarded as a separate parameter of the inductive description. In other cases, it can be encoded as a binary-valued variable. Practical examples with both these techniques are described. Extending further the use of multiple outputs, a representation for vectors of outputs is proposed, which is especially useful for handling compositions of parametric circuits. For example, the vector of read-port outputs from a parametric register file, can be composed as the input vector to a parametric adder. The compositions become trickier in the case of multiple parameters, e.g. when the ALU is parametric in the number of data lines (i), and the register file is parametric both in the number of data lines (i) and in the number of address lines (j).

Another interesting aspect of representing multiple parameter IBFs is in handling parameters for space and time simultaneously. This arises in practice while dealing with networks of processes or structurally-inductive ensembles of finite state machines in typical protocol verification applications. In the simplest case, the space and time parameters are independent, and the hardware system can be directly represented using the multiple parameter IBF framework. In other cases, it is possible to introduce dummy parameters and achieve canonicity within the new parametric domain. In yet others, fixed-length symbolic simulation yields convergence in the IBF representations, which corresponds to other work based on language containment for finite state automata [6, 131]. Thus, simultaneous parameterization in space and time can be readily incorporated within the unified IBF framework.

1.4.3 Applications for Formal Hardware Verification

The various applications of IBF methodology for formal hardware verification lie in exploiting the power of induction that the IBF schemata are based upon. A natural application is for functional verification of structurally-inductive hardware. By representing a correctness property

as an IBF formula, checking it with a proof by induction on the size parameter corresponds to an automatic tautology-check on the corresponding IBF representation. This not only allows complete automation of the induction proof, but also eliminates the heuristic search typically associated with more powerful proof-theoretic frameworks.

The LIF manipulation framework can also be used for behavioral verification of finite state sequential systems. For example, the canonicity property of LIF representations for FSM outputs can be directly used to show input/output equivalence of two FSMs with different number of states, and different state encodings. The relationship between an LIF representation and a standard DFA (deterministic finite state automaton) representation of a sequential function is further investigated in Chapter 5, where it is shown that an LIF representation is equivalent to a minimal *reverse* DFA, i.e. a DFA that accepts the input strings in reverse order. For many practical datapath circuits, the reverse DFA is exponentially more compact than the classic, forward DFA. This is especially useful since the LIF-based method for obtaining the reverse DFA does not require construction of the explicit forward DFA, unlike the existing methods for DFA reversal. Thus, the LIF method can be regarded as an alternative attack on the state explosion problem in such circuits.

Other interesting properties that can be symbolically captured using LIF manipulations include equivalence/containment of regular languages, the set of reachable states etc. The important point is that though symbolic state-space analysis is usually not viewed in terms of an induction proof, these manipulations capture the precise inductive characterization of such an analysis. Furthermore, the unified framework for handling structural and temporal parameters opens up possibilities for combining induction proofs in both domains. This includes both symbolic simulation of space-parametric circuits, as well as language containment between networks of automata.

In terms of the bigger picture, the verification applications of the IBF methodology strongly parallel those of the BDDs, as shown in Figure 1.4. At the heart of this relationship lies the correspondence between BDDs/tautology-checking on one hand, and the IBF schemata/induction proofs on the other. Just as BDDs have been widely used to perform both functional verification of combinational circuits and behavioral verification of finite state sequential systems, the IBF schemata can also be used to perform induction proofs for functional verification of (structurally-inductive) combinational circuits and state-space exploration of (temporally-inductive) sequential circuits. Typical examples of such applications are described in Chapter 6. Practical results from a prototype implementation of the methodology are also presented, along with a discussion of some practical issues.

The thesis concludes by highlighting the major contributions of this research. Along with its well-defined and stated objectives with regard to inductive verification, many aspects turned out to be interesting from a higher-level perspective also. Naturally, many questions still remain, some of which are targeted as potential directions for future research.

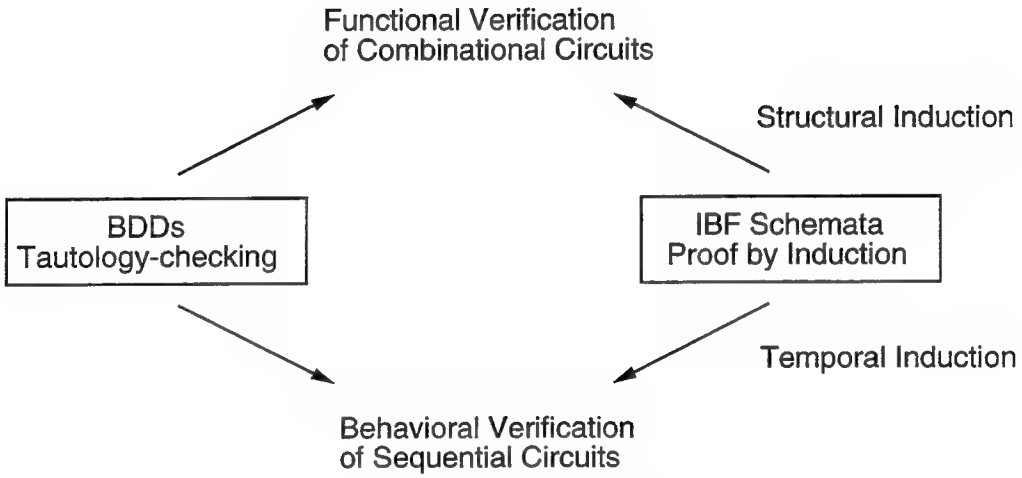


Figure 1.4: Relationship between Symbolic BDD Manipulation and IBF Methodology

Chapter 2

Inductive Boolean Functions

In this chapter, formal definitions and examples are provided for inductive Boolean functions (IBFs). Informally, IBFs are Boolean functions defined inductively in terms of formal parameters. Each such function denotes a family of standard Boolean functions, viz. those corresponding to instances with actual values of parameters. Naturally, the arguments of an IBF are also described parametrically in terms of the formal parameters. The exact form of the parametric definition of an IBF can vary, e.g. it may involve a dependence on the arguments only, or it may involve a combination of the arguments and other functions etc. Different forms of these definitions are regarded as leading to different *classes* of IBFs.

Two IBF classes are considered in this thesis – linearly inductive functions (LIFs), and exponentially inductive functions (EIFs). As the name suggests, the former class is intended to capture a linear, or serial, style of inductive definition. It corresponds most closely to iterative circuits. The latter class is intended to capture a divide-and-conquer, or parallel, style of inductive definition. It corresponds to typical tree circuits. Some further conditions are imposed with a view to obtaining efficient canonical representations. These representations, and symbolic manipulation algorithms are described in the next chapter.

Along with the formal definitions, some typical examples for each IBF class are also provided. The emphasis is on conveying a flavor of the inductive descriptions, without getting involved in the detailed notation (which is made more precise in the succeeding chapters). Pointers are also provided to existing work that is related to specific aspects of the IBF definitions.

2.1 Linearly Inductive Functions (LIFs)

2.1.1 Definition

Definition 1: A Linearly Inductive Function (LIF) f, is a parametric Boolean function, defined

inductively in terms of a parameter i as follows:

- Condition 1: For $i = 1$, the 1-instance of f (denoted f^1) is a Boolean combination (denoted $\mathcal{B}^1$) of the 1-instance inputs (denoted X^1),
  i.e. $f^1 = \mathcal{B}^1(X^1)$.
- Condition 2: For all $i > 1$, the i-instance of f (denoted f^i) is a Boolean combination (denoted $\mathcal{B}^i$) of only the i-instance inputs (denoted X^i) and some $(i-1)$-instance functions (denoted G^{i-1}), each of which is also an LIF,
  i.e. $f^i = \mathcal{B}^i\,(X^i, G^{i-1})$.
- Condition 3: For all $i, j > 1, \mathcal{B}^i \equiv \mathcal{B}^j$,
  i.e. $\mathcal{B}^i$ is isomorphic to $\mathcal{B}^j$ modulo substitution of parameters (f^i is related to X^i and G^{i-1} in the same manner as f^j is to X^j and G^{j-1}, respectively).

Note that Conditions 1 and 2 constitute the basis and the inductive parts of the parametric definition respectively, while Condition 3 imposes a further restriction on all parametric instances $i > 1$. Also, note that the definition only characterizes the relationship between the parametric instances of the IBFs and their arguments, without dictating the exact form of the parameterized arguments X^i themselves. For example, X^i could denote the i^{th} element of a vector X, or it could denote a subvector of X of length i. Both of these scenarios are allowed by the definition, and as formalized in a later chapter, are indeed used to handle circuits in practice. For now, X^i is used in the natural sense in the examples discussed in the following section.

2.1.2 Examples of LIFs

2.1.2.1 Structural Induction

By using circuit size as a parameter of the inductive description, LIFs can can naturally capture induction in the structural domain. Typically, the circuit structure can be described inductively in terms of the size parameter, and the circuit output for the i^{th}-instance of the circuit is expressed as the i-instance of an LIF.

Example 1: Consider a parametric description of a ripple carry adder of size i, with parameterized data inputs a, b and a carry input c_in, as shown in Figure 2.1. It consists of a linear array of i identical cells, with each cell j receiving inputs a^j and b^j, and the leftmost cell also receiving c_in. There are two outputs of the circuit – sum and $carry$. The basis values of these outputs (sum^1 and $carry^1$), produced by the leftmost adder cell, are Boolean combinations of the basis inputs (a^1, b^1, and c_in). The i-instance values of these outputs (sum^i and $carry^i$), produced by the i^{th} adder cell, are Boolean combinations of the i-instance inputs (a^i, b^i) and

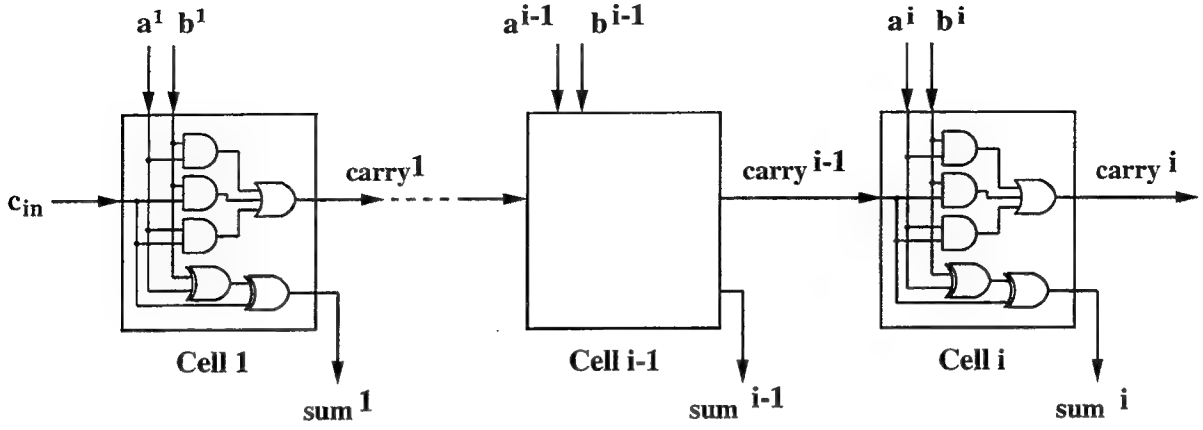


Figure 2.1: Parametric Description of a Ripple Carry Adder

the $(i-1)$-instance of the function $carry$ ($carry^{(i-1)}$). Thus, the parametric functions sum and $carry$ are LIFs. Their inductive definitions can be stated as:

for $i = 1, sum^1 = a^1 \oplus b^1 \oplus c\_in$
for $i > 1, sum^i = a^i \oplus b^i \oplus carry^{i-1}$

where the LIF $carry$ is defined as:

for $i = 1, carry^1 = (a^1 \wedge b^1) \vee ((a^1 \vee b^1) \wedge c\_in)$
for $i > 1, carry^i = (a^i \wedge b^i) \vee ((a^i \vee b^i) \wedge carry^{i-1})$

Example 2: Consider a serial parity circuit of size i, with parameterized input x, as shown in Figure 2.2. Again, it can be regarded as an array of i cells, each cell j $(1 \leq j \leq i)$ receiving input x^j. There is one output $parity$, such that its basis value ($parity^1$) is simply the same as the basis input (x^1), and its i-instance value ($parity^i$) is obtained from the i-instance input (x^i) and the $(i-1)$-instance value of the same function ($parity^{(i-1)}$). In terms of the LIF description:

for $i = 1, parity^1 = x^1$
for $i > 1, parity^i = x^i \oplus parity^{(i-1)}$

Example 3: As a final example, consider a comparator circuit of size i, which compares parameterized inputs x and y, and has an output geq which is true when $x \geq y$. Its inductive structure is as shown in Figure 2.3. The basis value (geq^1) is a function of the basis inputs (x^1 and y^1). The i-instance value (geq^i) is a function of the i-instance inputs (x^i,y^i), and the $(i-1)$-instance of the same function ($geq^{(i-1)}$). Its LIF description is as follows:

for $i = 1, geq^1 = (x^1 \wedge \neg y^1) \vee \neg(x^1 \oplus y^1)$
for $i > 1, geq^i = (x^i \wedge \neg y^i) \vee (\neg(x^i \oplus y^i) \wedge geq^{(i-1)})$

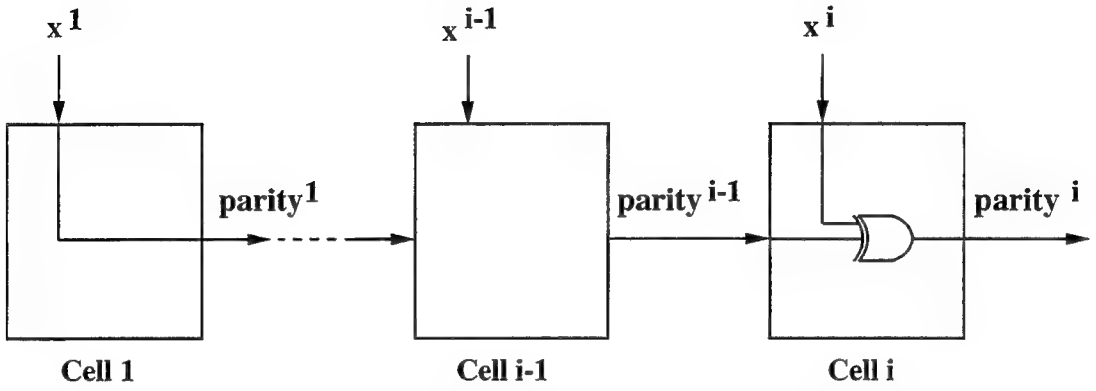


Figure 2.2: Parametric Description of a Serial Parity Circuit

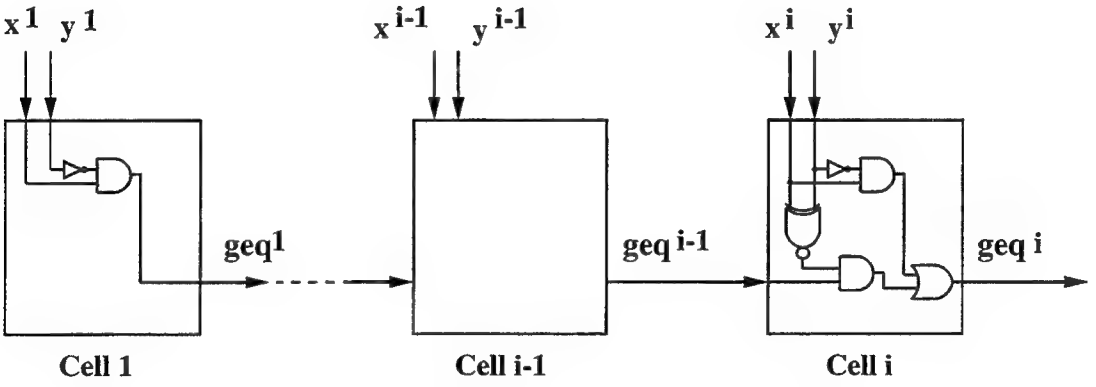


Figure 2.3: Parametric Description of a Comparator

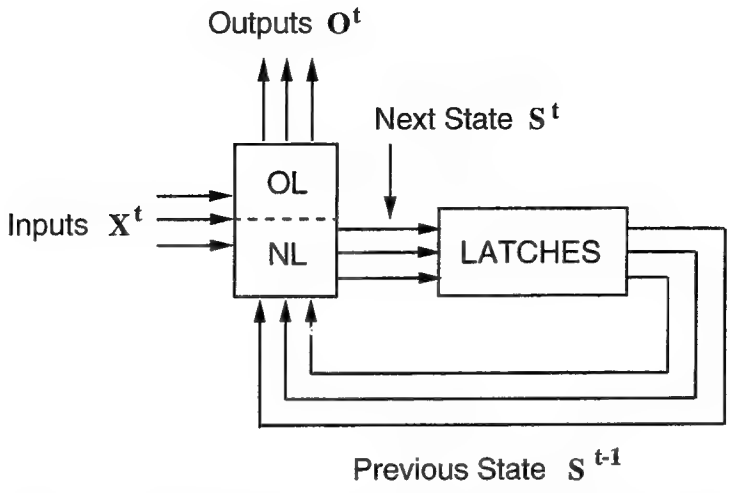


Figure 2.4: Mealy Machine Model of a Finite State Machine

Note that in all examples, a designated set of basis variables (X^1) is used for defining the basis instance of an LIF. A basis variable can be either the basis instance of a parameterized variable, e.g. a^1 and b^1 of the ripple carry adder; or a non-parameterized variable, e.g. c_in of the ripple carry adder. On the other hand, for $i > 1$, the set of variables used for defining the i-instance of an LIF (X^i) are i-instances of parameterized variables. In a later chapter, conditions are described under which the i-instance of an LIF can also depend upon non-parameterized variables. Such a need often arises in order to capture the dependence on "control" variables, e.g. variables used to select the operation of an ALU. Special conditions are enforced in order to ensure canonicity of the resulting representation. For now, only those examples will be described where the i-instance of an LIF depends upon i-instances of parameterized variables only.

2.1.2.2 Temporal Induction

By regarding the induction parameter as denoting time, LIFs can also capture induction in the temporal domain. In fact, all finite state sequential systems can be viewed as computing functions serially over time, and all sequential functions can be expressed as LIFs. In particular, consider a Mealy machine model of a deterministic finite state machine (FSM), as shown schematically in Figure 2.4. Formally, a Mealy machine model is represented as a six-tuple $(Q, \Sigma, \Delta, T, O, q)$, where

- Q is a finite set of states
  (encoded by binary variables $S = [s_1, s_2, \ldots, s_n]$ in the figure)

- Σ is a finite input alphabet
  (encoded by binary variables $X = [x_1, x_2, \ldots, x_m]$ in the figure)
- Δ is a finite output alphabet
- T is the next-state transition function, $T : S \times \Sigma \rightarrow S$
  (shown as next-state-logic NL in the figure)
- O is the output function, $O : S \times \Sigma \rightarrow \Delta$
  (shown as output-logic OL in the figure)
- q_0 is the initial state, $q_0 \in Q$

Given the symbolic state vector $S = [s_1, s_2, \ldots, s_n]$ that represents the states, and the symbolic input vector $X = [x_1, x_2, \ldots, x_m]$ that represents the inputs, the T and O functions can be regarded as vectors of LIFs. T consists of transition functions for each state bit s_i $(1 \leq i \leq n)$, and O consists of the output functions for each output bit o_i $(1 \leq i \leq k)$ as follows:

T: for $t > 1, s_i^t = \mathcal{B}_{s_i}(X^t, S^{t-1})$
O: for $t > 1, o_i^t = \mathcal{B}_{o_i}(X^t, S^{t-1})$

The values of these functions at $t = 1$ represent the initial state bits and initial output bits respectively.

Example 4: As a concrete example of a finite state machine, consider the state transition diagram shown in Figure 2.5 for a standard 2-bit counter. The symbolic state vector consists of two state bits $S = [s1, s2]$ (labeled inside each state). The input vector consists of a single input $X = [e]$, which is used to enable counting up. The output vector consists of two bits $O = [o1, o2]$. (Each transition in the diagram is marked with the corresponding input/output label, and the initial state is marked with an arrow.) The LIFs for the state transition functions (T) and the output functions (O) can be expressed as follows:

for $t = 1, s_1^1 = o_1^1 = 0$
for $t > 1, s_1^t = o_1^t = (\neg e^t \wedge s_1^{t-1}) \vee (e^t \wedge \neg s_1^{t-1})$

for $t = 1, s_2^1 = o_2^1 = 0$
for $t > 1, s_2^t = o_2^t = (\neg e^t \wedge s_2^{t-1}) \vee (e^t \wedge (s_2^{t-1} \oplus s_1^{t-1}))$

2.1.3 Related Work

The class of LIFs most closely resembles iterative networks, i.e. networks composed of a cascade of identical circuits or cells [93]. Depending on whether the cell consists of combinational

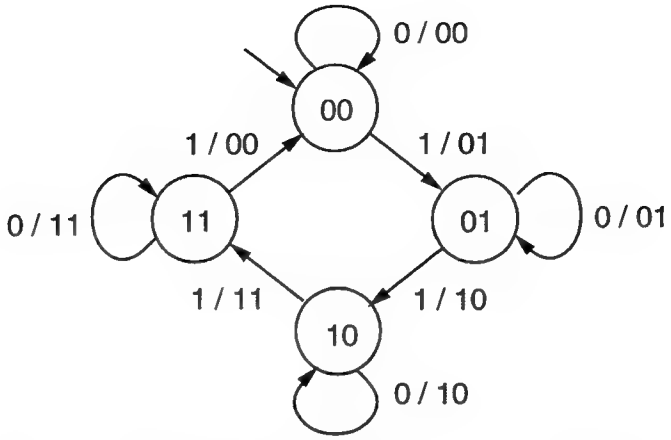


Figure 2.5: State Transition Diagram for a 2-bit Counter

logic or sequential logic, the network is classified as either a combinational network or a sequential network, respectively. There are further categories based on the number of dimensions of the cell array, as well as the number of directions in which signals flow through the network. In comparison to non-structured networks of similar size complexity, the iterative structure of these networks makes them simpler to analyze and to synthesize. On a more practical note, it is also easier to manufacture and maintain them.

Iterative networks were introduced by Keister, Ritchie, and Washburn in the context of relay switching networks [91]. The space-time analogy between unilateral combinational networks and sequential systems was observed by Huffman [82], and its correspondence to Moore's finite state machine model[1] was presented by McCluskey [107]. This allowed the use of finite state automata techniques for the analysis and synthesis of such networks. A detailed study of iterative networks, and techniques for deciding some interesting properties about them, are described in the classic work by Hennie [74].

The class of single-dimensional LIFs, as defined in the previous sections, corresponds closely to iterative functions of *unidimensional unilateral combinational iterative networks*. However, there are several reasons for defining the class of LIFs anew. Apart from the minor technical differences in the definitions itself, the analysis of LIFs also differs considerably from the classic work on iterative networks, and is summarized as follows:

- The LIF definition allows for a richer set of basis instances for iterative functions, i.e. the basis definition for $i = 1$ can be different from the inductive definition for $i > 1$. Typically, iterative networks consist of identical cells for $i = 1$ and for $i > 1$. The only

[1] In the Moore machine model, outputs are a function of the state only, i.e. $OF : S \rightarrow O$. Rest of the model is identical to the Mealy machine model described in this section.

difference allowed is in defining the inputs to the cell for $i = 1$, called the "boundary conditions".

- The focus in this thesis is on *functions* that can be defined iteratively, and not on the structure of the underlying network. Traditionally, the studies of iterative networks have focused very strongly on the structural properties of networks, e.g. stability, regularity etc. The main emphasis in this thesis, instead, is on studying the functional relationship between the outputs and inputs of the iterative network.

- The emphasis on functions is motivated by the desire to provide a framework for their symbolic manipulation, and to allow for a dynamic system of functions where more functions may be added as more manipulations are performed. Such manipulations are very useful for tasks such as formal verification. Existing studies of iterative networks deal with static network structures only, and it is not immediately clear how those techniques can be extended to deal incrementally with a dynamically changing set of functions.

- An essential component of the symbolic LIF manipulation framework is an efficient representation for the iterative functions. Most studies of iterative networks use a standard "flow table" (state transition table) for representing the input/output behavior of a single cell. Furthermore, some pre-assigned encoding is used for the inter-cell signals. By contrast, the LIF representation provides an encoding-independent canonical representation of the iterative functions. This provides an easy handle for symbolically manipulating the functions, rather than working with the operational semantics of a flow table.

- Finally, the goal of extending the existing BDD techniques for Boolean functions, makes it relatively natural to start from a class of *inductive* Boolean functions. The inductive definitions also make it conceptually easier to incorporate the reasoning by induction into the symbolic manipulation itself. Though the standard iterative network techniques also use a proof by induction at the meta-theoretical level, it is not operationally incorporated into the function representations. Furthermore, not many attempts have been made at automating the formal reasoning about such systems.

2.2 Exponentially Inductive Functions (EIFs)

2.2.1 Definition

Informally, inductive instances of EIFs are formed from Boolean combinations of lower-instance functions applied to the left and right halves of the parametric input vector. This

makes the EIF class very useful for handling binary tree circuits. In this class, an i-instance input V^i is regarded as a vector of 2^i scalar inputs ($V^i = [x_1, x_2, \ldots, x_{2^i}]$).

Definition 2: An Exponentially Inductive Function (EIF) f is a parametric Boolean function, defined inductively in terms of a parameter i as follows:

- Condition 1: For $i = 0$, the 0-instance of f (f^0) is a Boolean combination ($\mathcal{B}^0$) of 0-instance inputs (V^0), each 0-instance input consisting of a scalar input,
  i.e. $f^0 = \mathcal{B}^0(V^0)$.

- Condition 2: For all $i > 0$, the i-instance of f (f^i) is a Boolean combination ($\mathcal{B}^i$) of $(i-1)$-instances of three functions e, g and h, each of which is an EIF,
  $\mathcal{B}^i$ is restricted such that,
  e^{i-1} is applied to the left half of the i-instance input (denoted V_L^i)), and
  g^{i-1} and h^{i-1} are applied to the right half of the i-instance input (denoted V_R^i),
  i.e. $f^i = \mathcal{B}^i(e^{i-1}(V_L^i), g^{i-1}(V_R^i), h^{i-1}(V_R^i))$.

- Condition 3: For all $i, j > 0$, $B^i \equiv B^j$.

Note that the main difference between the class of LIFs and the class of EIFs is reflected in Condition 2 of their definitions. In LIFs, the inductive i-instance function ($i > 1$) is allowed to explicitly depend upon the i-instance inputs. However, in EIFs, the inductive i-instance function ($i > 0$) does not *explicitly* depend upon the i-instance inputs. Rather, it is a restricted Boolean combination of $(i-1)$-instance EIFs, which *implicitly* depend upon the $(i-1)$-instance inputs. (In this regard, note that V_L^i and V_R^i denote halves of the i-instance input V^i, and are therefore regarded as $(i-1)$-instance inputs.) In the EIFs, the explicit dependence on inputs is allowed only in the basis case ($i = 0$). (The reason for choosing $i = 0$, and not $i = 1$, as the basis case is briefly explained towards the end of this chapter.)

Furthermore, unlike the generality in LIFs, where an i-instance function can be defined in terms of *any* Boolean combination of $(i-1)$-instance functions, EIFs are restricted to use only certain Boolean combinations of $(i-1)$-instance functions, as indicated in Condition 2. This restriction can be understood in the following sense. Imagine a circuit that computes f^i by partitioning the inputs into halves, and by recombining results from the two subcircuits. Condition 2 for EIFs implies that this partitioning takes place recursively at each level of induction, such that recombination at that level potentially involves the transfer of *only one bit* of information from the left half to the right half. Note that this does not preclude auxiliary computation in the left half, in order to produce that one bit of information. However, it does require that this auxiliary computation should also satisfy the same property.

As described in more detail in the next chapter, these restrictions are motivated primarily by the desire to obtain efficient canonical representations for the class of EIFs. Furthermore, different aspects of these restrictions can be relaxed for handling practical circuits. In the next chapter, a

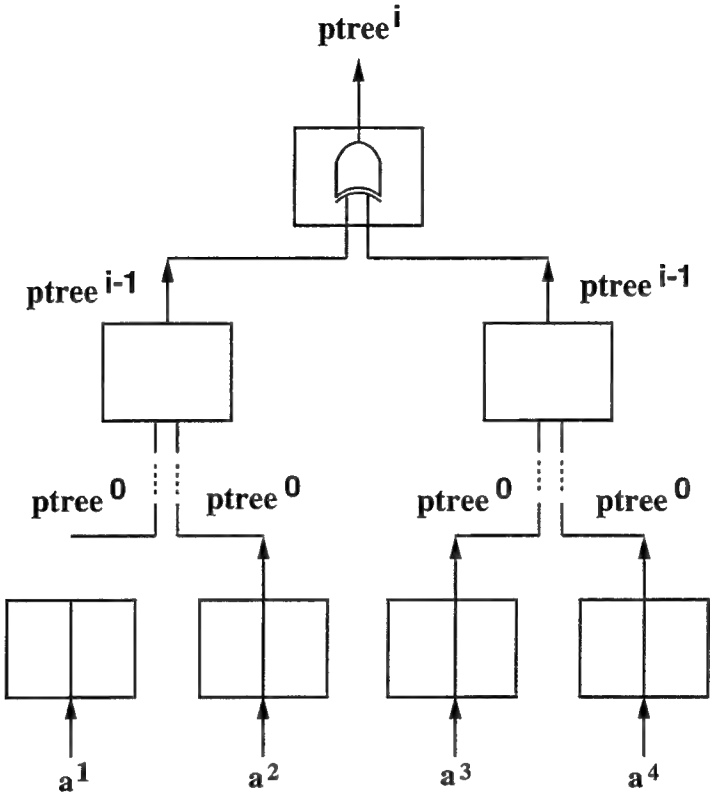


Figure 2.6: Parametric Description of a Tree Parity Circuit

general framework for representation of IBFs is described, such that both the LIF representation and the EIF representation can be regarded as special cases of the general IBF representation. Other classes of IBFs can be potentially explored within that spectrum. For now, examples of practical circuits that can be naturally captured as EIFs are described in the following section.

2.2.2 Examples of EIFs

Example 5: Consider an i-instance tree parity circuit as shown in Figure 2.6. It has a parameterized input a, represented as a 2^i-bit vector $a[2^i]$. The parity output function $ptree^i$ can be expressed as an EIF as follows:

for $i = 0, ptree^0(a[1]) = a[1]$;
for $i > 0, ptree^i(a[2^i]) = ptree^{i-1}(a[2^i]_L) \oplus ptree^{i-1}(a[2^i]_R)$

Note that the e, g, and h functions of the inductive i-instance EIF definition, refer to the same

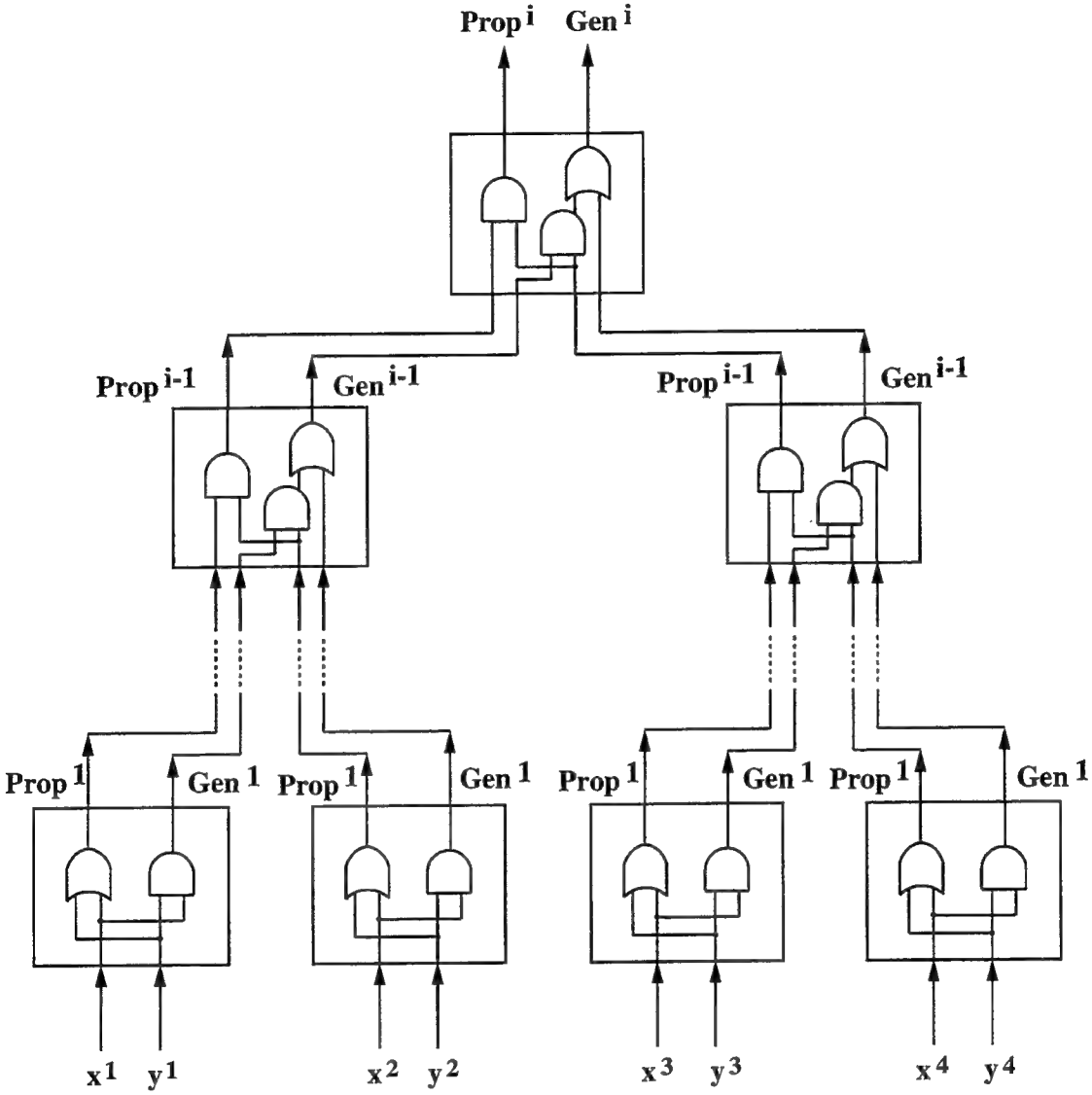


Figure 2.7: Parametric Description of a Tree Carry Lookahead Adder

function P in this example.

Example 6: Consider next an i-instance tree carry lookahead adder as shown in Figure 2.7. The parameterized inputs x and y are each represented as a vector of 2^i inputs – $x[2^i]$ and $y[2^i]$ respectively. The carry propagate ($prop$) and carry generate (gen) functions can be defined as:

for $i = 0, prop^0(x[1], y[1]) = x[1] \vee y[1]$

for $i > 0, prop^i(x[2^i], y[2^i]) = prop^{i-1}(x[2^i]_L, y[2^i]_L) \wedge prop^{i-1}(x[2^i]_R, y[2^i]_R)$

for $i = 0, gen^0(x[1], y[1]) = x[1] \wedge y[1]$
for $i > 0, gen^i(x[2^i], y[2^i]) = (gen^{i-1}(x[2^i]_L, y[2^i]_L) \wedge prop^{i-1}(x[2^i]_R, y[2^i]_R)) \vee$
$gen^{i-1}(x[2^i]_R, y[2^i]_R)$

Note that the definition of $prop$ is already in the form of the EIF definition (where e, f, and h in the inductive i-instance definition refer to the same function $prop$). However, for gen^i ($i > 0$), the function definition can be rewritten as follows:

for $i > 0, gen^i(x[2^i], y[2^i]) = (\neg gen^{i-1}(x[2^i]_L, y[2^i]_L) \wedge gen^{i-1}(x[2^i]_R, y[2^i]_R)) \vee$
$(gen^{i-1}(x[2^i]_L, y[2^i]_L) \wedge (gen^{i-1}(x[2^i]_R, y[2^i]_R) \vee prop^{i-1}(x[2^i]_R, y[2^i]_R)))$

Next, the Boolean combination $gen^i \vee prop^i$ is renamed as an auxiliary function $temp^i$, which replaces the last two terms in the previous expression:

for $i > 0, gen^i(x[2^i], y[2^i]) = (\neg gen^{i-1}(x[2^i]_L, y[2^i]_L) \wedge gen^{i-1}(x[2^i]_R, y[2^i]_R)) \vee$
$(gen^{i-1}(x[2^i]_L, y[2^i]_L) \wedge temp^{i-1}(x[2^i]_R, y[2^i]_R))$

The above definition now conforms to the standard EIF definition (where e, g, and h of the inductive i-instance definition refer to gen, gen, and $temp$, respectively.) Note that the auxiliary function $temp$ is also an EIF, defined as follows:

for $i = 0, temp^0(x[1], y[1]) = x[1] \vee y[1]$
for $i > 0, temp^i(x[2^i], y[2^i]) = (\neg temp^{i-1}(x[2^i]_L, y[2^i]_L) \wedge gen^{i-1}(x[2^i]_R, y[2^i]_R)) \vee$
$(temp^{i-1}(x[2^i]_L, y[2^i]_L) \wedge temp^{i-1}(x[2^i]_R, y[2^i]_R))$

2.2.3 Related Work

Interest in recursive tree circuits started with efforts to minimize circuit delay in comparison to iterative circuits [21, 139, 144, 151]. Typically, the delay for outputs in an iterative circuit is linear in the number of cells n. The most celebrated example is that of a ripple carry adder, where an n-bit adder has an $O(n)$ worst-case delay.

General techniques have been provided for generating a tree type of network (called a combinational iterative tree) from the state transition descriptions of cells of any linear network [145]. The delay for such a tree network is thereby reduced to $O(\log n)$, while the size complexity is maintained at $O(n)$. The general technique relates directly to the well-known examples for an adder circuit, including the tree carry lookahead adder [57, 117], and the conditional sum adder [138]. In a way, the technique can be viewed as converting a tail-recursive (iterative) procedure to a binary recursive one.

Since the study of binary tree circuits has taken place mostly in the context of minimizing propagation delays, their equivalence to corresponding iterative circuits has been the primary focus of attention in the formal verification area [149, 150]. However, not much work exists on directly reasoning about such circuits. By providing a separate class of EIFs for handling binary tree circuits, the required symbolic machinery is made available to reason about them directly. It is also interesting, in this context, to see which techniques are shared by both LIFs and EIFs. Such techniques can potentially form a core methodology for handling IBFs in general.

As shown in the next chapter, the IBF methodology provides for canonical representation of an IBF *only within its own class*. For example, the LIF representation for a ripple carry adder is *not* the same as the EIF representation for a tree carry lookahead adder, even though the final outputs of these circuits are functionally equivalent. One of the interesting directions for future work, is to explore how representations for different IBF classes can be manipulated in a deliberative automatic manner, in order to prove such equivalences. At a minimum, this would require the machinery for symbolic manipulation of integer expressions, which is outside the scope of this thesis. The potential for proving such equivalences is the prime motivation for defining $i = 1$ as the basis-instance of an LIF, and $i = 0$ as the basis-instance for an EIF. The symbolic integer manipulation is simplified when it is required, for example, to prove that an i-instance EIF is functionally equivalent to a 2^i-instance LIF, for all $i \geq 0$.

Chapter 3

IBF Schemata

This chapter describes schemas for the two classes of IBFs identified in the previous chapter. Each schema consists of a canonical function representation and algorithms for symbolic manipulation of functions in that class. For both classes, the representations are extensions of Binary Decision Diagrams (BDDs) [22], made popular by Bryant for representation of Boolean functions. The immense success of BDDs is primarily due to the their canonicity property, i.e. for a fixed ordering of variables, a BDD representation of a Boolean function is canonical. Furthermore, though of NP-hard complexity, the BDD operations for performing symbolic Boolean function manipulation are efficient in practice.

The important property of IBF representations presented here, is that they too are canonical with respect to a fixed ordering of the parametric variables. However, unlike the BDD representations where the canonicity property can be syntactically ensured right from the start, IBF representations require building intermediate forms starting from the user definitions. An explicit equality check between intermediate forms is needed in order to obtain final canonical forms. Thus, the entire system gradually progresses from intermediate forms towards final canonical forms for all IBFs. Furthermore, symbolic IBF manipulation algorithms are guaranteed to preserve canonicity of those representations that are already in the final form. These algorithms for building canonical IBF representations, and for symbolic IBF manipulation, are efficient in practice for a wide range of hardware systems, as demonstrated in the next few chapters. The focus in this chapter is on the details of the representations and algorithms, along with an analysis of the worst-case complexity. A description of the LIF schema can also be found in an associated paper [69].

A brief review of BDDs is provided next, as an introduction to the terminology used for the IBF schemata.

3.1 Review of Binary Decision Diagrams

The following material is summarized from Bryant's original paper on BDDs [22], with some terminology from later papers [18, 27].

3.1.1 BDD Representation

A Binary Decision Diagram represents a Boolean function as a rooted, directed acyclic graph (dag). Each non-terminal node v is labeled by a Boolean variable $var(v)$, and has two outgoing edges to nodes $hi(v)$ and $lo(v)$, corresponding to the variable being true or false, respectively. Each terminal node represents a Boolean constant, and is labeled 1 (true) or 0 (false).

A Boolean function f_v is associated with each node v of a dag, such that:

- if v is a terminal node,
  $f_v = 1$ or $f_v = 0$, depending on the Boolean constant that v represents.
- if v is a non-terminal node labeled by variable x,
  $f_v = \neg x \wedge f_{lo(v)} \vee x \wedge f_{hi(v)}$, where
  $f_{lo(v)}$ denotes the function associated with node $lo(v)$, and
  $f_{hi(v)}$ represents the function associated with node $hi(v)$.

Thus, the value of a Boolean function associated with a node v can be obtained simply by following the assignment of variables along a directed path from v to a terminal node, and the corresponding Boolean constant denotes the value of the function.

An Ordered BDD is a BDD with the additional constraint that all directed paths in the DAG consist of non-terminal nodes in an *ascending order*, specified as a total ordering on the set of variables. This variable ordering is fixed for the BDD representation of all Boolean functions in the system. Though the BDD algorithms work correctly for an arbitrary variable ordering, typically their complexity, and the size of the resulting BDDs, is sensitive to the particular ordering chosen.

The DAG representing an Ordered BDD can be further reduced by using the following transformation rules, each of which preserves the function represented by each node in the dag:

- Remove duplicate terminals, i.e. retain only one terminal node per Boolean constant.
- Remove duplicate non-terminals, i.e. retain only one non-terminal node v for a given triple $< var(v), hi(v), lo(v) >$.
- Remove redundant tests, i.e. remove a node v where $lo(v) = hi(v)$.

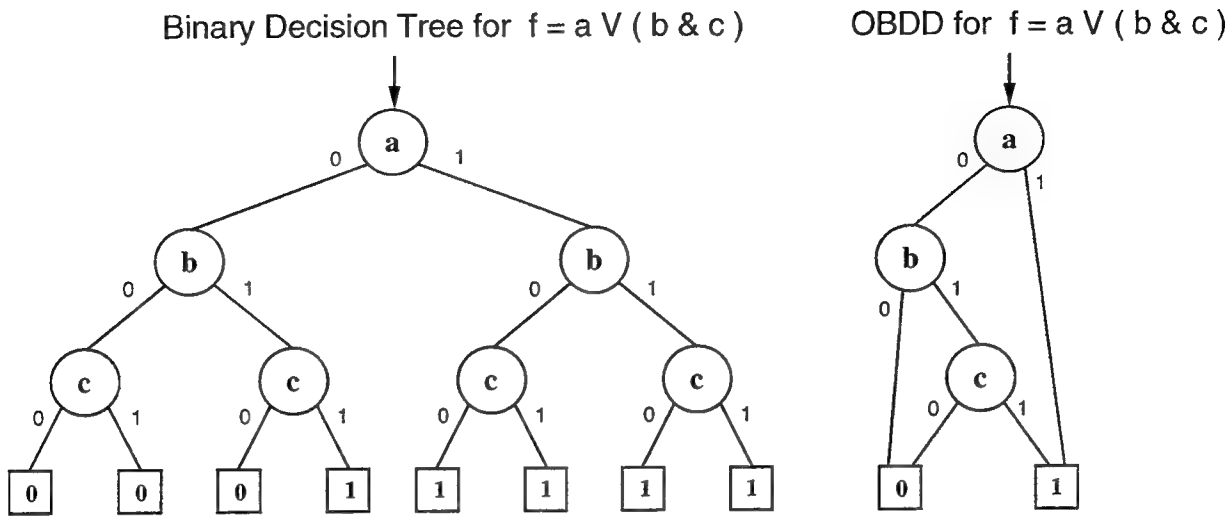


Figure 3.1: Binary Decision Tree and OBDD Representation

These rules ensure that there are no isomorphic subgraphs or redundant nodes in a BDD dag. Also, they can be applied repeatedly until no exbdd3.iv further reduction is possible. Typically, the term "OBDD" is used to refer to a maximally reduced Ordered BDD. Note that the *ordered* and *maximally reduced* properties distinguish an OBDD representation from the more common binary decision tree representation of a Boolean function. As an example, the binary decision tree and OBDD representations for a function $f = a \vee (b \wedge c)$ are shown in Figure 3.1, for the variable ordering $a < b < c$. The notation used in these figures is:

- DAG edges implicitly point from top to bottom.
- Non-terminal nodes are shown as circles labeled by variables.
- The outgoing edge from node v to node $lo(v)$ is labeled 0, and that to node $hi(v)$ is labeled 1.
- Terminal nodes are shown as boxes labeled with Boolean constants.

3.1.2 Boolean Operations using OBDDs

Bryant has provided graph-based algorithms for performing symbolic Boolean operations using the OBDD representations [22]. Most of these rely on the Shannon's expansion of a Boolean function f for a variable x, defined as follows:

$f = \neg x \wedge f|_{x=0} \vee x \wedge f|_{x=1}$

In the above equation, the term $f|_{x=0}$ is called the *Restriction* (or *Cofactor*) of f with respect to $x = 0$, and represents the value of f when x is assigned a value 0. (Similarly $f|_{x=1}$ is called the restriction of f with respect to $x = 1$). With the OBDD representation, the restriction operation is easily implemented as picking up either the $lo(v)$ or the $hi(v)$ child of any non-terminal node v with $var(v) = x$, followed by reduction of the resulting dag.

Bryant also provides a general *Apply* operation with two Boolean functions and a binary operator as arguments. It can be used to obtain the standard algebraic operations such as disjunction, conjunction, parity etc. Since these operators commute with the Shannon's expansion of a Boolean function with respect to a variable, the *Apply* operation is implemented as a recursive algorithm on the structure of the argument OBDDs, where the non-terminal nodes provide the splitting variable. The recursion is terminated at the OBDD terminal nodes, by supplying the appropriate Boolean constant result. In practice, a memoization technique is used to keep track of each recursive operation triple (the two OBDD arguments, and the operator), such that no multiple calls are issued for the same triple. Also, reduction transformations are applied locally after each recursive call, making it unnecessary to further reduce the top-level result dag. These enhancements bound the overall complexity of the *Apply* algorithm, and the size of the resulting OBDD, to be some polynomial function of the argument OBDDs.

The *Apply* operation is very useful in practice. It enables building the OBDD representations for a network of circuit elements, by successively applying the corresponding Boolean operations, starting from simple OBDD representations for symbolic Boolean variables. The other option – that of starting from a complete binary decision tree representation of a Boolean function, followed by graph reduction – would defeat the very purpose of trying to avoid the exponential-sized truth tables.

Other useful Boolean operations can be defined in terms of restriction and the algebraic operations.

- *Composition*, where a function f is substituted for variable x of function g, is defined as

  $$g|_{x=f} = \neg f \wedge g|_{x=0} \vee f \wedge g|_{x=1}$$

- *Existential / Universal Quantification* of a function f, with respect to a variable x, is defined as

  Existential: $\exists x.f = f|_{x=0} \vee f|_{x=1}$
  Universal: $\forall x.f = f|_{x=0} \wedge f|_{x=1}$

The complexity of algorithms for these operations is typically polynomial (linear to quadratic) in the size of the OBDDs for the argument functions. As mentioned earlier, the correctness of these Boolean operation algorithms holds regardless of the variable ordering used. However, the OBDD sizes, and thereby the practical complexity of such manipulation, is frequently sensitive to the variable ordering used.

3.1.3 Canonicity Property of an OBDD

Bryant has shown that, given a fixed variable ordering, an arbitrary Boolean function can be represented by a unique (up to isomorphism) OBDD [22]. Checking equivalence of two Boolean functions is done simply by checking that their OBDDs are isomorphic. Thus, the circuit equivalence problem, which is known to be of NP-hard complexity [60], also corresponds to an isomorphism check on the OBDDs. Similarly, tautology-checking of a Boolean formula is operationally simplified by checking that its OBDD is isomorphic to the Boolean constant node '1'. Satisfiability of a Boolean formula translates to checking that its OBDD is not isomorphic to the Boolean constant node '0'. The complexity of these checks is reflected in the work needed to obtain the OBDD representations, because in a typical implementation the OBDD isomorphism check is a constant-time operation, due to availability of a strong (syntactic) canonical form [18].

3.1.4 Efficiency Issues: Variable Orderings and Inherent Complexity

Though Boolean function manipulation using OBDDs is of NP-hard complexity, the efficiency in practice depends on the size of the OBDD representations encountered in a particular application. As mentioned earlier, the choice of a variable ordering usually has a significant effect on the OBDD sizes. For the same function, a bad variable ordering can result in an exponential-sized BDD, whereas a good ordering may produce a linear-sized one. However, finding an optimal ordering is itself an NP-hard problem. Therefore, several heuristics have been developed for choosing a satisfactory ordering in practice [58, 105]. These have contributed to the success of OBDD-based symbolic manipulation in a wide variety of CAD applications, such as synthesis, testing, simulation and formal verification of digital systems [27]. On the other hand, pushing the application size boundaries (an ever-present demand), still requires the user's intuition about the underlying functions. A relatively recent alternative uses an automatic scheme, where the variable ordering is dynamically changed when a pre-defined memory limit is reached [132]. Though it involves a performance penalty, its transparency from the user is a considerable advantage.

Apart from the effect of variable ordering on the size of OBDD representations, there is also an issue of inherent complexity. Lower bounds and upper bounds have been obtained for several important classes of Boolean functions. For example, symmetric functions have linear to quadratic size representations, and are insensitive to variable orderings [27]. On the other hand, the middle bits of an integer multiplier have exponential-sized BDD representations regardless of the variable ordering used [26]. Some other classes of functions, especially for structured hardware organizations, relate strongly with the IBF representations. These are described in the next section.

3.2 LIF Schema

In this section, the schema for the representation and symbolic manipulation of LIFs is described. First, inductive extensions to the OBDD representation are motivated. This is followed by addressing the all-important issue of canonicity. The algorithms designed for obtaining canonical LIF representations, including the basic framework for symbolic LIF manipulation, are described in detail. This is accompanied with a complexity analysis and a comparison with related work. The salient features of a practical implementation for an LIF manipulation package are also described.

3.2.1 LIF Representation

Given that each i-instance of an LIF is a Boolean combination of the i-instance inputs and some $(i-1)$-instance LIFs, it is natural to use an inductive extension of an BDD to capture an i-instance LIF. One such candidate is called a *Layered BDD (LBDD)*, with an associated parameter called *level*. Intuitively, an i-level LBDD captures the layer of i-instance variables in the BDD representation of an i-instance LIF. However, additional machinery is needed in order to capture *all* instances of an LIF. The progression of these representations is described in detail in this section, starting with the Layered BDD.

3.2.1.1 Layered BDD (LBDD) Representation

A Layered BDD (LBDD) is a parametric version of a BDD, with a *level* parameter. Accordingly, the variables for the LBDD are also parameterized, i.e. they denote parameterized inputs.

Definition 3: The inductive definition for an i-level LBDD is as follows:

- A 1-level LBDD is a standard BDD with 1-instance variables.
- An i-level LBDD, as shown in Figure 3.2, is a BDD-like DAG with non-terminal nodes representing i-instance variables, and with terminal nodes that consist of either:
  - Boolean constants, or
  - pointers to $(i-1)$-level LBDDs.

Note from the figure that an i-level LBDD is much like a standard BDD on i-instance variables except for its terminal nodes. This allows easy modifications of standard BDD algorithms, where the modifications are targeted only at the terminal nodes.

A Boolean function f_v can be associated with node v of an i-level LBDD in much the same way as for a BDD:

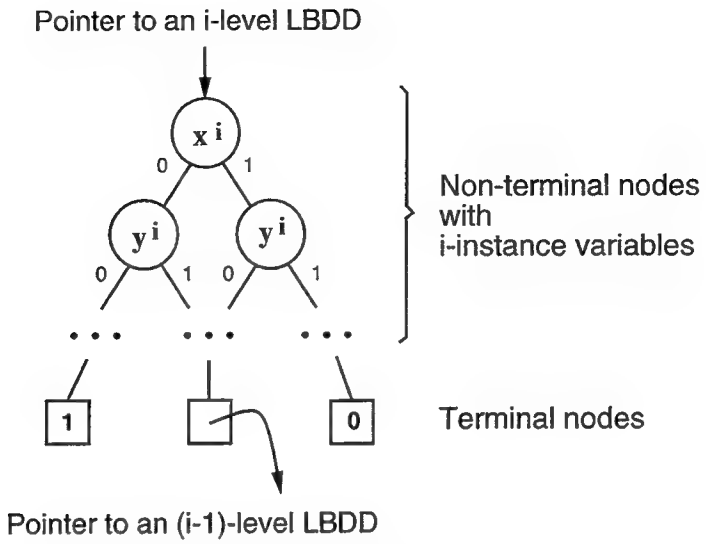


Figure 3.2: An i-level LBDD Representation

- if v is a terminal node representing a Boolean constant,
  $f_v = 1$ or $f_v = 0$, depending on the Boolean constant value that v represents.
- if v is a terminal node with a pointer to an $(i-1)$-level LBDD node w,
  $f_v = f_w$, where f_w is well-defined by an inductive argument on i.
- if v is a non-terminal node representing the i-instance of a parameterized variable x,
  $f_v = \neg x^i \wedge f_{lo(v)} \vee x^i \wedge f_{hi(v)}$, where
  $f_{lo(v)}$ denotes the function associated with the $lo(v)$ node, and
  $f_{hi(v)}$ denotes the function associated with the $hi(v)$ node.

As in the case of BDDs, a total ordering on variables is used to obtain Ordered LBDDs. Recall that both parameterized and non-parameterized variables are allowed in the LIF definitions. Therefore, a total ordering includes both kinds of variables. However, the ordering for parameterized variables is understood to apply to all i-instances, thereby leading to the *same ordering between all i-instances of those variables*. For example, consider a variable ordering $a < x < y < z < b$, where a, b are non-parameterized variables, and x, y, z are parameterized variables. This implies that for any $i \geq 1$, $a < x^i < y^i < z^i < b$. This requirement is not critical for the argument regarding canonicity of Ordered LBDDs (given in the next section), but becomes critical for what follows later on.

Transformation rules similar to those for BDDs, are used to obtain maximally reduced Ordered LBDDs. The only modification is with respect to the terminal nodes, where two terminal nodes are regarded as duplicates if and only if:

- they denote the same Boolean constant, or
- they point to the same $(i-1)$-level *maximally reduced* Ordered LBDD.

Note that the "maximally reduced" requirement is essential in the second case above. Note also that it is well-defined by an inductive argument on the level parameter i.

3.2.1.2 Canonicity Property of an LBDD

Theorem 1: For a fixed variable ordering, f^i for an LIF f, $i \geq 1$, can be represented in a canonical form using an i-level maximally reduced Ordered LBDD.

Proof: The theorem can be proved by a simple induction on i as follows.

Basis case for $i = 1$: f^1 is a Boolean function on the 1-instance inputs X^1. Therefore, f^1 can be represented as a standard OBDD on X^1. By definition, this is a 1-level Ordered LBDD, and is canonical due to the OBDD canonicity property.

Induction hypothesis: Let the theorem be true for some $i = k$.

Induction step: Now consider the case for $i = k+1$. From the LIF definition, f^{k+1} is a Boolean combination of X^{k+1} and G^k. Note also that a Boolean combination of k-instances of LIFs is the k-instance of another LIF, i.e. the set of k-instances of LIFs is closed under Boolean operations. By the induction hypothesis, each of these k-instances can be represented in a canonical form by a maximally reduced k-level Ordered LBDD. Therefore, f^{k+1} can be represented as an OBDD with X^i variables as non-terminal nodes, and with terminal nodes that are either Boolean constants, or point to k-level Ordered LBDDs. By definition, this DAG is a $(k+1)$-level Ordered LBDD. Due to the modified transformation rules above, and due to the canonicity property of an OBDD and the given terminal nodes, the maximally reduced DAG is also canonical. ■

3.2.1.3 Relationship between an Ordered LBDD and an OBDD

Since the Ordered LBDD is defined as an inductive extension of an OBDD, an i-level Ordered LBDD representation for f^i bears a strong relationship with a standard OBDD representation for f^i. In order to see this clearly, we first define the notion of a *consistent* variable ordering, relating an ordering on parameterized variables, to an ordering on all instances of the parameterized variables.

Definition 4: An ordering β on all instances of parameterized variables is defined to be *consistent* with an ordering ρ on parameterized variables, if:

- for all $i \geq 1$, β preserves the ordering between i-instances of all parameterized variables according to ρ.
- for all $i > 1$, for all parameterized variables, i-instance variable $<$ $(i-1)$-instance variable in β.

Note that if f is regarded as a multiple-output function (each output being $f^i, i \geq 1$), then β captures the natural variable ordering for a multiple-root OBDD representation for f [22].

It is now easy to see that an Ordered LBDD representation (with a given ordering ρ), is a way of partitioning a monolithic OBDD representation (with an ordering β that is consistent with ρ) into layers. This is demonstrated in Figure 3.3 for the $Carry$ output function of a ripple carry adder (Example 1). Note from the figure that each layer i ($i > 1$) of the OBDD representation refers only to the i-instance variables a^i and b^i, and a pointer into layer $(i-1)$ represents the contribution of $Carry^{i-1}$ to $Carry^i$. For $i = 1$, no pointers are needed, and the terminal nodes are simply Boolean constants. Thus each layer i of the OBDD representation is captured by an i-level Ordered LBDD.

3.2.1.4 Function Descriptor Representation

A canonical i-level Ordered LBDD representation for an i-instance of an LIF, does not resolve the problem of representing *all* instances of the LIF. The useful potential of the LBDD representation lies in exploiting Condition 3 in the LIF definition (Definition 1, Chapter 2), which states that for $i, j > 1$, the Boolean combination functions are isomorphic, i.e. $\mathcal{B}_i \equiv \mathcal{B}_j$. For the LBDD representation, this condition implies that the DAG structure of the i-level Ordered LBDD is isomorphic to the dag structure of the j-level Ordered LBDD, *modulo substitution of parameters*. Note that this substitution affects the instances of parameterized variables represented by the non-terminal nodes, as well as the instances of LIFs pointed to by the terminal nodes.

Thus, the representation for all i-instances of an LIF, for $i > 1$, can be captured by maintaining a parameter substitution semantics on only one LBDD structure. For $i = 1$, the representation of the 1-instance of an LIF is already available in the form of a standard OBDD. Therefore the entire LIF (all i-instances, $i \geq 1$) can be represented as follows:

Definition 5: An LIF f is represented as a *Function Descriptor (FD)* consisting of:

1. an OBDD on 1-instance variables, which represents the 1-instance of f — called the *Basis BDD* for f, and
2. a parametric i-level Ordered LBDD with substitution semantics, which represents all i-instances of f for $i > 1$ — called the *Linearly Inductive BDD (LIBDD)* for f.

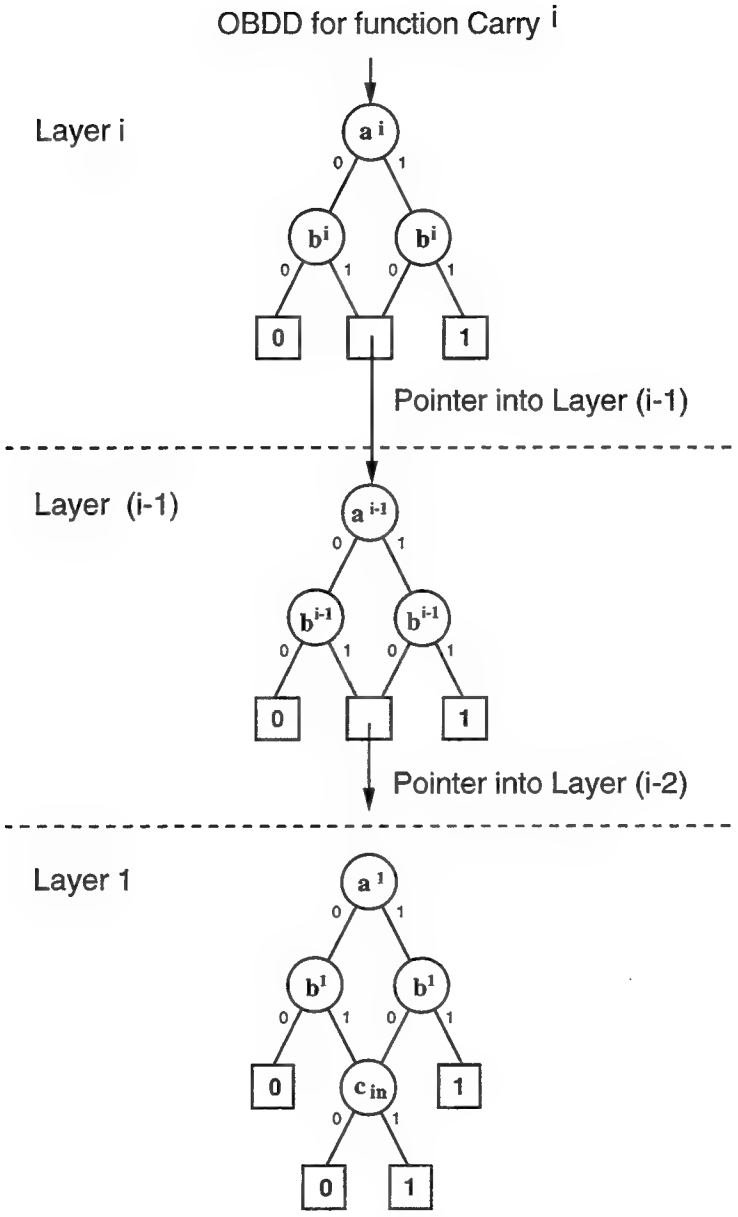


Figure 3.3: Partitioning of an OBDD into Layers

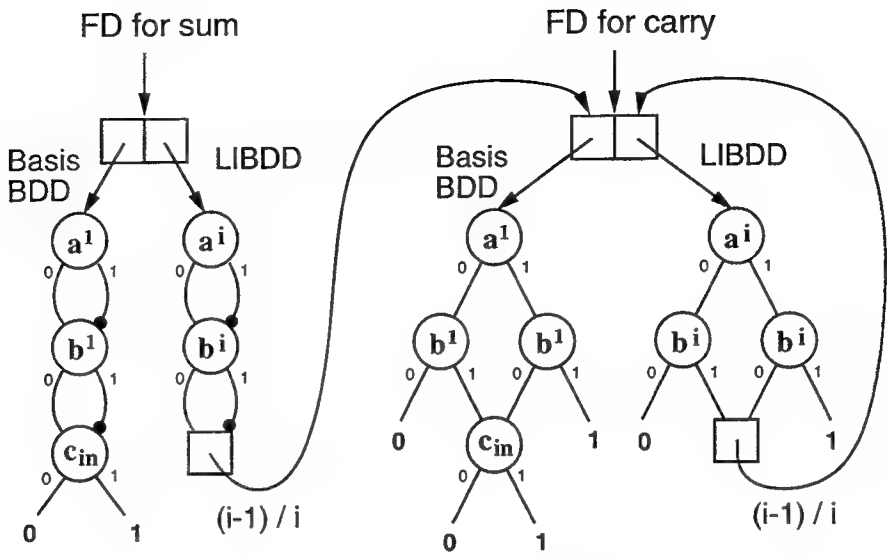


Figure 3.4: LIF Representation for a Ripple Carry Adder

The substitution semantics on the LIBDD dictate that its terminal nodes can now point to FDs. The reason can be understood as follows. Note that an LIBDD represents i-instance of an LIF, and its terminal nodes can point to j-instances of LIFs, where $j = i - 1$. If the LIBDD represents all $i > 1$, it requires that $j \geq 1$. Thus, the LIBDD terminal node should point to the representation for all j-instances, $j \geq 1$, i.e. the corresponding FD. To summarize, the non-terminal nodes of the LIBDD represent i-instance variables as before, but the terminal nodes can point to FDs, and implicitly use the parameter substitution $(i-1)/i$ (to be read as "substitute $(i-1)$ for i in the LIBDD").

For example, the FDs for LIFs sum and $carry$ of the ripple carry adder (Example 1), are shown in Figure 3.4, where the parameter substitution on pointers from LIBDD terminal nodes to FDs is indicated as '$(i-1)/i$'. In this figure (and in the rest of the thesis), a dark circle on an edge in the BDD representation denotes a negative edge attribute [12, 18], i.e. it denotes complementation of the function pointed to by the edge. For example, in the Basis BDD of the LIF sum, the value of the function for $(a^1 = 1, b^1 = 0, c\_in = 1)$ is 0 because of the negation on the 1-edge from a^1. On the other hand, the value of the function for $(a^1 = 1, b^1 = 1, c\_in = 1)$ is 1 because of the double negation on the 1-edges from a^1 and b^1. In general, the use of negative edge attribute can reduce the BDD storage requirements by half, for a minimal overhead in symbolic manipulation. Other edge attributes have also been proposed and used in various implementations [18, 103, 116].

As in the case of Ordered LBDDs, a total ordering on all variables – parameterized, as well as non-parameterized – is used for the Basis BDD and the LIBDD. Though the same ordering is

typically used for both the Basis BDD and the LIBDD, it is not a technical requirement. The only requirement is that all Basis BDDs in the system should follow the same variable ordering, and similarly all LIBDDs in the system should follow the same variable ordering. In addition, note that since the LIBDD uses a parameter substitution mechanism for all $i > 1$, it is essential that all i-instances of parameterized variables should have the same ordering for $i > 1$. This is the reason for originally specifying the variable ordering on parameterized variables, and not on their i-instances. For the rest of this thesis, it is assumed that the Basis BDD and the LIBDD are ordered according to a given variable ordering, unless otherwise specified.

3.2.2 Canonicity of LIF Representations

So far the LIF representation has been shown to capture the definition of an LIF in graphical form – the Basis BDD for the basis instance definition, and the LIBDD for the inductive instance definition. In this section, the method for making these representations canonical is described. Since each i-instance of an LIF, f^i, depends implicitly on all variables X^j, $1 \leq j \leq i$, this method can be viewed as capturing in canonical form, the functional relationship between f^i and the sequence of inputs X^j, $1 \leq j \leq i$.

In order for the FD representation to be canonical, the canonicity of both the Basis BDD and the LIBDD need to be ensured. Since the Basis BDD is a standard OBDD, it can be made canonical by construction through use of any of the standard techniques [18]. As for the LIBDD, note that the canonicity argument for the i-level LBDD (Section 3.2.1.2) does not apply directly, because it relies inductively on the canonicity of the $(i-1)$-level LBDD, which is now represented by the same LIBDD. However, going back yet another step in the same argument, recall that if canonicity of the terminal nodes of an i-level LBDD can be independently ensured, then rest of the DAG can be made canonical by following the BDD transformation rules. Thus, *the main task is to establish canonicity of the LIBDD terminal nodes, some of which can potentially point to FDs.*

Two main problems need to be addressed in order to accomplish this task. They are best illustrated by the following example.

Example 7: Consider three user-defined LIFs f, g, and h:

for $i = 1, f^1 = x^1$	for $i > 1, f^i = x^i \wedge g^{i-1}$
for $i = 1, g^1 = x^1$	for $i > 1, g^i = x^i \wedge h^{i-1}$
for $i = 1, h^1 = x^1$	for $i > 1, h^i = x^i \wedge (f^{i-1} \vee g^{i-1})$

The first problem is that since the inductive definitions of these LIFs are mutually-recursive, a bottom-up approach starting from canonical terminal nodes cannot be used for building the LIBDD, as is done for a standard BDD. Therefore, the strategy used is to start from *tentative*

LIBDDs and FDs – *tentative* in the sense that they are not necessarily canonical to start with. This allows tentative LIBDDs to be obtained in a bottom-up manner, where tentative FDs are regarded as simple place-holders for pointers from LIBDD terminal nodes. Subsequently, the tentative FDs are made canonical by checking for distinctness using an explicit equality check.

The second problem is demonstrated by the definition of h^i, which involves a Boolean combination of $(i-1)$-instance LIFs f^{i-1} and g^{i-1}, as allowed by Condition 2 of the LIF Definition. Recall that all references to an $(i-1)$-instance LIF are captured through the terminal node of an LIBDD (pointer from the terminal node to the FD corresponding to that LIF). Therefore, the same mechanism should be used to handle *a Boolean combination* of $(i-1)$-instance functions also. Fortunately, any Boolean combination of $(i-1)$-instance functions is another $(i-1)$-instance function. Thus, a *new* LIF can be generated to denote the required Boolean combination, and any reference to this combination is directed to the new LIF. (The method used to obtain its FD is described in the next section.) Note that the system has to now keep track of all user-defined LIFs, as well as the newly generated LIFs denoting Boolean combinations.

These problems can be handled by conceptually separating the task into the following two phases:

- **Generation Phase**
  In this phase, the task is to build tentative FDs bottom-up, starting from the user-given definitions of LIFs in the system. Also, new LIFs are generated to denote all required Boolean combinations of user-defined LIFs.

- **Comparison Phase**
  Once all required tentative FDs are available, they are made canonical by performing an explicit comparison against all existing members of the canonical set. A tentative FD is added to the set of canonical FDs if and only if it is distinct.

3.2.2.1 Generation Phase

A tentative FD is associated with each user-defined LIF in the system. As mentioned earlier, each required Boolean combination of user-defined LIFs is renamed to be a *new* LIF, with an associated tentative FD. The method used to build tentative FDs for both user-defined LIFs and newly generated LIFs is described in the following subsection.

Building the Tentative FDs

For the user-defined LIFs, the Basis BDD and the tentative LIBDD follow the definitions provided by the user. The Basis BDD is obtained in the usual way by successively applying appropriate Boolean operations to standard BDDs that denote basis variables. For the tentative

LIBDD, a similar approach is used to build the structure bottom-up, by successively applying Boolean operations to:

- LIBDDs that denote parameterized variables, and
- LIBDDs consisting of a single terminal node with a pointer to a tentative FD, denoting the $(i-1)$-instance of the corresponding LIF.

For a newly generated LIF which denotes a Boolean combination of two user-defined LIFs, the FD is obtained by applying the corresponding Boolean operation to the FDs of the user-defined LIFs. In other words, the Basis BDD and the tentative LIBDD for the new LIF are obtained by applying the corresponding Boolean operations to the Basis BDDs and tentative LIBDDs of the user-defined LIFs.

Thus, Boolean operations on the Basis BDDs and the tentative LIBDDs are needed to obtain the FDs for both the user-defined LIFs and the newly generated LIFs. For the Basis BDDs, the Boolean operations are performed by the standard BDD *Apply* operations on the argument Basis BDDs [22], shown here in Figure 3.5. In the shown procedure, the auxiliary function *minimum_order* returns the argument which is minimum in the global variable ordering, and *BDD_find* is either finds an existing BDD node with the required triple $< var(v), hi(v), lo(v) >$, or creates a new one if none exists.

For the LIBDD, the Boolean operations are simple extensions of the standard BDD *Apply* operation, which modify the termination of recursion, as shown in Figure 3.6. In the case where LIBDD terminal nodes denote Boolean constants, the result is also a Boolean constant (depending on the desired Boolean operation). However, if these nodes contain a pointer to a tentative FD, they are handled differently. (If only one argument v is a terminal node with a pointer to a tentative FD, the other argument is recursively traversed till both become terminal. This is typically implemented by ensuring $var(v)$ to be the last in the global variable ordering.) Semantically, a Boolean operation on two such terminal nodes denotes a Boolean combination of two $(i-1)$-instance LIFs. This is handled in the same way as a Boolean combination of two user-defined LIFs, i.e. a new LIF (and a tentative FD) is generated to denote the required Boolean combination (using the auxiliary procedure *LIF_Apply*, described in detail later). Again, the tentative FD is used as a place-holder for the pointer from the terminal node of the result LIBDD. (A handle to such an LIBDD terminal node is obtained by using the auxiliary procedure *LIBDD_terminal_node*.)

For Example 7, the tentative FDs (TFDs) for f, g and h are shown in Figure 3.7. (In these figures, the first slot of a TFD is understood to point to the Basis BDD, and the second slot to the LIBDD. Also, the parameter substitutions on the pointers from LIBDD terminal nodes to TFDs is implicit.) Note that the LIBDD structures for f and g mirror the user-definitions. For h, the Boolean combination of f and g is renamed to a new LIF p, defined $p^i = f^i \vee g^i$, such

```
procedure BDD_Apply(f, g, < op >)

    if f is a Boolean constant or g is a Boolean constant
        return (f < op > g)

    top_var = minimum_order(var(f), var(g))

    if (top_var == var(f))
        f_1 = hi(f), f_0 = lo(f)
    else
        f_1 = f, f_0 = f

    if (top_var == var(g))
        g_1 = hi(g), g_0 = lo(g)
    else
        g_1 = g, g_0 = g

    result_1 = BDD_Apply(f_1, g_1, < op >)
    result_0 = BDD_Apply(f_0, g_0, < op >)
    result = BDD_find(top_var, result_1, result_0)

    return(result)
end
```

Figure 3.5: BDD *Apply* Operation

```
procedure LIBDD_Apply(f, g, < op >)

    if f is a Boolean constant or g is a Boolean constant
        return (f < op > g)

    if f is a terminal node with pointer to f_FD
        if g is a terminal node with pointer to g_FD
            new_FD = LIF_Apply(f_FD, g_FD, < op >)
            return(LIBDD_terminal_node(new_FD))

    top_var = minimum_order(var(f), var(g))

    if (top_var == var(f))
        f_1 = hi(f), f_0 = lo(f)
    else
        f_1 = f, f_0 = f

    if (top_var == var(g))
        g_1 = hi(g), g_0 = lo(g)
    else
        g_1 = g, g_0 = g

    result_1 = LIBDD_Apply(f_1, g_1, < op >)
    result_0 = LIBDD_Apply(f_0, g_0, < op >)
    result = LIBDD_find(top_var, result_1, result_0)

    return(result)
end
```

Figure 3.6: LIBDD *Apply* Operation

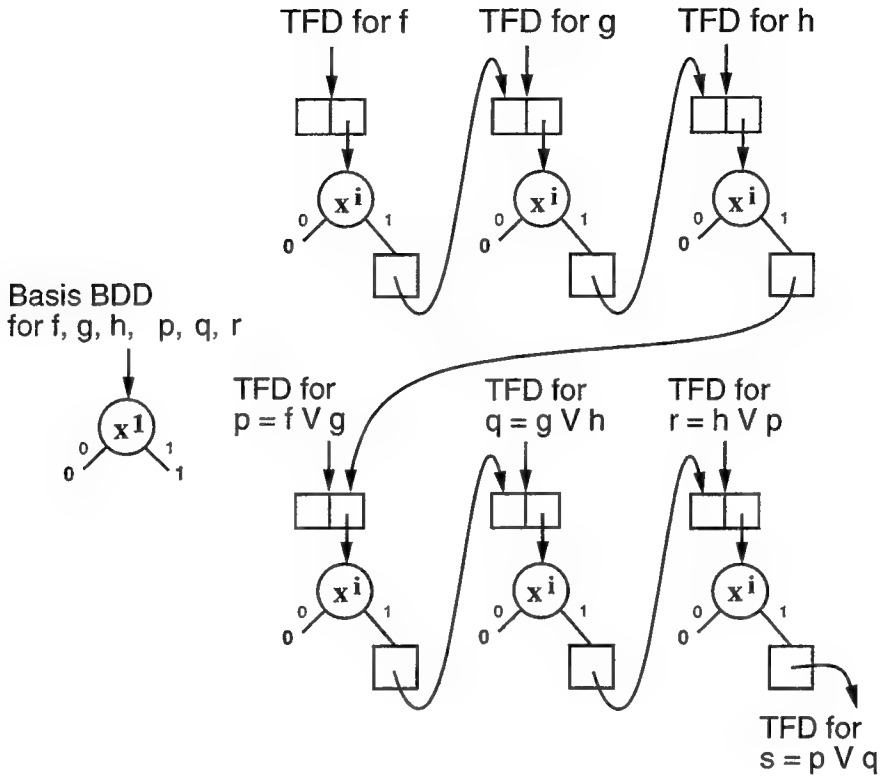


Figure 3.7: TFDs for Example 7

that terminal node of the tentative LIBDD for h points to the TFD for p. Next, the TFD for p is obtained by using *Apply* on the Basis BDDs and LIBDDs for f and g. In this process, a new LIF is generated, defined $q^i = g^i \vee h^i$ (arising from the Boolean combination on the right terminal nodes of the LIBDDs for f and g). Again, the TFD for q is generated, which requires another new LIF r, defined $r^i = h^i \vee p^i$. The TFD for r potentially leads to yet another new LIF s, defined $s^i = p^i \vee q^i$, as shown in the same figure.

Note that each new LIF (p, q, r, and s) denotes a Boolean combination of two previously defined LIFs. However, the renaming is such that an infinite series of new LIFs seems to be needed, implying that the Generation Phase would never terminate. Such a situation should definitely be avoided. Let us examine again what the latest LIF s denotes.

$$
\begin{aligned}
s^i &= p^i \vee q^i && \text{by definition of } s \\
&= (f^i \vee g^i) \vee (g^i \vee h^i) && \text{by substitution for } p \text{ and } q \\
&= f^i \vee g^i \vee h^i && \text{by simplification}
\end{aligned}
$$

$= (f^i \vee g^i) \vee h^i$ by regrouping
$= p^i \vee h^i$ by substituting in p
$= r^i$ by definition of r

Thus, s does *not* denote a new LIF; in fact, it denotes the same Boolean combination of user-defined LIFs as r. For efficiency reasons itself, it would be important to not generate *redundant* combinations during generation of new LIFs. However, in this case, it is crucial to avoid generating redundant combinations, in order to guarantee that the LIF Generation Phase terminates. This is described in the next section.

Termination of the LIF Generation Phase

The key idea for avoiding redundant Boolean combinations of user-defined LIFs is to maintain canonical forms with respect to user-defined LIFs. In other words, when new LIFs are generated, their equivalence classes with respect to user-defined LIFs are maintained. Note that this is different from equivalence with respect to the underlying variables (which is the final goal). For example, consider two user-defined LIFs a and b defined in terms of variables v and w. Define two new Boolean combination LIFs c and d, where $c = a \wedge b$, and $d = \neg(a \oplus b)$. Clearly, the two LIFs c and d are *not* equivalent with respect to the LIFs a and b. However, c and d may be equivalent with respect to the underlying variables v and w, e.g. when $a = v \vee w$ and $b = \neg(v \wedge w)$. This simple example demonstrates that equivalence with respect to the underlying variables does not imply equivalence with respect to the user-defined LIFs. However, the reverse implication is always true, i.e. equivalence with respect to user-defined LIFs implies equivalence with respect to the underlying variables. Thus, in the Generation Phase, the LIF equivalence classes based on user-defined LIFs are *conservative* with respect to the underlying variables. These classes are subsequently made more exact in the Comparison Phase, described in the next section.

A straight-forward way to maintain canonicity with respect to user-defined LIFs is to use a separate Boolean space, called the meta-space. In the meta-space, a meta-variable is assigned to each user-defined LIF. Therefore, BDDs on these meta-variables, called meta-BDDs, denote arbitrary Boolean combinations of user-defined LIFs. In fact, each LIF in the system – user-defined, as well as newly generated – can be associated with a corresponding meta-BDD in the meta-space. The canonicity property of meta-BDDs is now exploited, whereby a new LIF is generated if and only if it corresponds to a distinct meta-BDD in the meta-space. This ensures that all equivalent Boolean combinations of user-defined LIFs are renamed to the same new LIF. This also guarantees termination of the LIF Generation Phase, since for a finite number of user-defined LIFs (u), there are a finite number of distinct Boolean combinations (2^{2^u}), and hence a finite number of meta-BDDs.

As mentioned earlier, for each new LIF generated to represent a distinct Boolean combination of user-defined LIFs, its tentative FD is obtained by using the corresponding Boolean operations

on the FDs of the argument LIFs. This process can also be viewed as using *parallel function composition*, where the FD corresponding to a new meta-BDD is obtained by simultaneously composing in FDs for all the meta-variables in the meta-BDD.

Getting back to Example 7, the meta-space for the LIFs in the system is shown in Figure 3.8, Part (a). Each of the user-defined LIFs – f, g, and h – is associated with a meta-variable. The newly generated LIFs – p, q, r, and s – are associated with meta-BDDs representing the corresponding Boolean combinations of the user-defined LIFs. Note that while p, q, and r represent distinct meta-BDDs, s represents the same meta-BDD as r. Therefore, the LIF s is not required, and r should be used instead. Recall from Figure 3.7, that s was needed for the terminal node of the tentative LIBDD for r. Thus, this terminal node now points back to r, as shown in Figure 3.8, Part (b). Now that the tentative FDs for all LIFs in the system have been obtained, the LIF Generation Phase is complete.

3.2.2.2 Comparison Phase

In this phase, the tentative FDs obtained during the Generation Phase are made canonical with respect to the underlying variables. Since the tentative FDs represent classes of LIFs that are already equivalent with respect to the user-defined LIFs, the task is to further combine those classes that are equivalent with respect to the underlying variables also.

An explicit set of canonical FDs is maintained, in a manner similar to a standard scheme for BDDs [18]. The idea is to convert each tentative FD to a canonical one, either by finding an equivalent FD in the canonical set, or by adding it as a new member to the set if it is unequal to all existing members. The difference from the standard BDD scheme is that while the BDD equivalence check is purely syntactic, the equivalence check for FDs is an explicit, non-syntactic check, described later in this section. Whenever a tentative FD is added to the canonical set, all the tentative FDs pointed to by its LIBDD terminal nodes are also made canonical.

Continuing with Example 7, assume that the set of canonical FDs is initially empty. So pick a candidate, say the TFD for f, and add it the canonical set. Now, all TFDs pointed to by its LIBDD terminal nodes need to be made canonical. From Figure 3.8, note that f's LIBDD terminal node points to the TFD for g. However, before adding the TFD for g also to the canonical set, its distinctness from the current set needs to be checked, which contains only the FD for f. Thus, an equivalence check on the TFDs for g and f is required. From Figure 3.8, is easy to see that the TFD for g is equivalent to the TFD for f *provided* the TFD for h is equivalent to the TFD for g. In other words, $g = f$ if $h = g$.

Such dependencies are represented in the form of a directed graph called a Comparison-graph, defined as follows:

Definition 6: A Comparison-graph is a directed graph on nodes denoting pairs of FDs, where

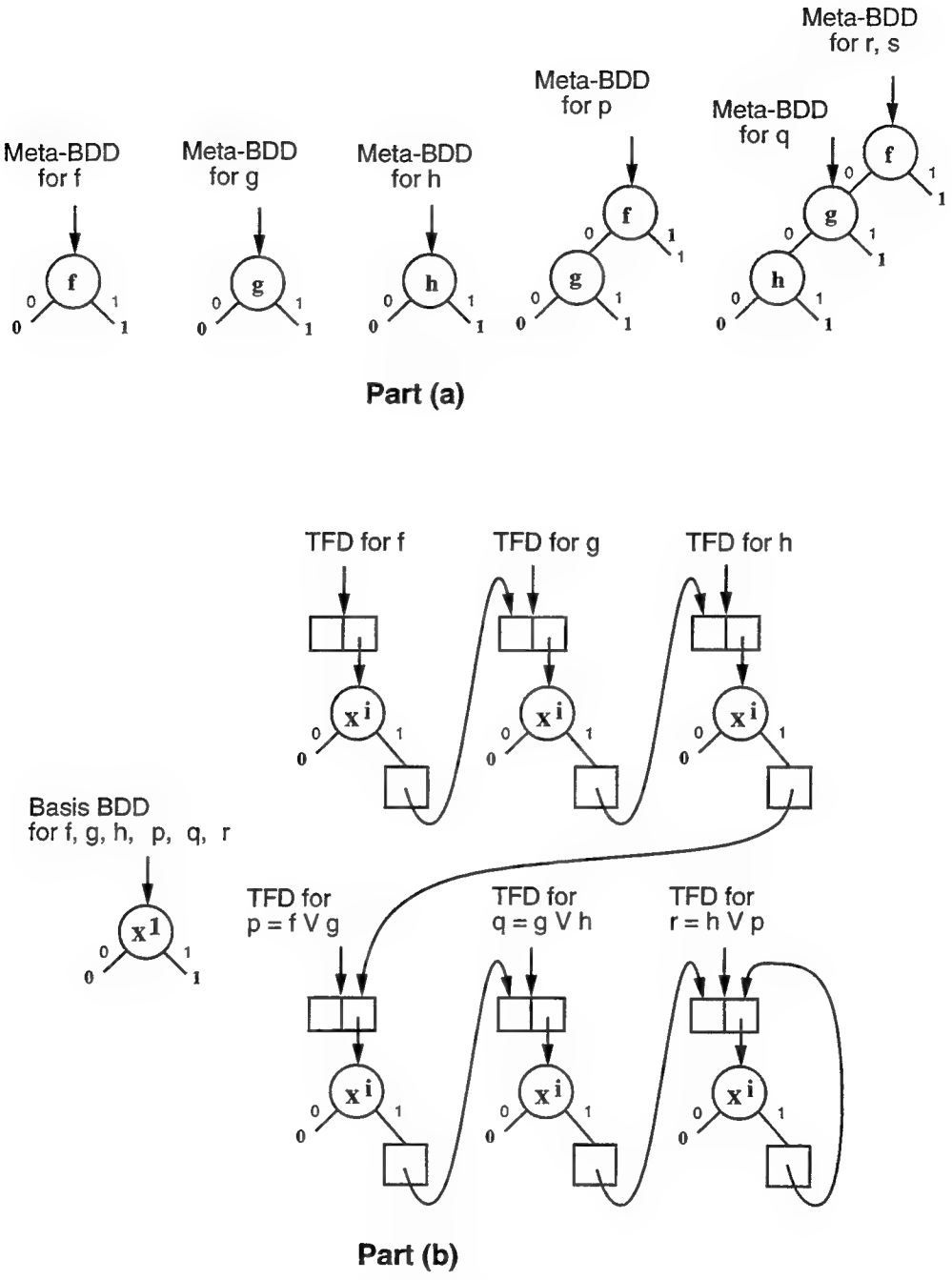


Figure 3.8: Meta-BDDs and TFDs for Example 7

- a node (p, q) represents the equivalence of the FDs p and q.
- an edge exists from a node (p, q) to a node (s, t), if the equivalence of the FD pair (p, q) depends on the equivalence of the FD pair (s, t), i.e.
  if $s \neq t \Rightarrow p \neq q$.

A separate table of results is maintained to record results of equivalence checks for all pairs of FDs. Nodes in the Comparison-graph are labeled 'True/False' according to these results. In Figure 3.9, the high-level algorithm $EQ(p, q)$ is shown, which is used for checking equivalence of FDs p and q using a Comparison-graph.

A new Comparison-graph is constructed for each top-level call to $EQ(p, q)$, but it uses previously established results recorded in the table (Steps 2 and 3). Step 5 can be implemented as a standard BDD algorithm on the recursive binary structure of an LIBDD, where Step 5(b) may recursively lead to new equivalence checks on pairs of FDs. Though it is not specifically required, the LIBDD traversal and the recursive checking of the LIBDD terminal nodes is accomplished in a single depth-first traversal over the pair of LIBDDs, which corresponds to a depth-first exploration of nodes in the Comparison-graph. As soon as the check for any of the LIBDD terminal node pairs fails (Step 5(e)), the traversal is abandoned after labeling the Comparison-graph nodes appropriately, i.e. rest of the Comparison-graph is not even explored.

The termination condition for the recursion in Step 5 involves an interesting interplay between the mutually-recursive and inductive aspects of LIF definitions, and is captured by the following assertion.

Theorem 2: The equivalence of FDs corresponding to a node (p, q) in a Comparison-graph is true if and only if there is no directed path from (p, q) to a node labeled 'False'.

Proof: Assume there is a directed path from (p, q) to a 'False' node. Note that a node labeled 'False' denotes a pair of FDs that are unequal, either due to the result of a previous computation (Step 3) or due to non-equivalence arising out of comparison of their Basis BDDs (Step 4) or LIBDDs (Step 5). According to the semantics of our Comparison-graph, a directed path from (p, q) to a 'False' node represents a chain of dependencies which culminates in a pair of unequal FDs. By transitivity of the implication relation, the FDs p and q are also unequal. (In fact, the length i of the shortest path to a 'False' node can be used in a counter-example facility to provide a particular instance of the respective LIFs which demonstrates the non-equivalence.)

For a proof in the other direction, assume there is no directed path from (p, q) to a 'False' node. In order to prove that the equivalence $p = q$ is true, we need to prove that *all* dependencies for node (p, q) are satisfied, i.e. the dependencies allow $p^i = q^i$ for all $i \geq 1$. This is proved by showing that each dependency chain, corresponding to a directed path from (p, q), is satisfied. Note that in the depth-first exploration of the Comparison-graph, a node can be left unexplored only if a directed path from it reaches a 'False' node (Step 5(b)). Since there is no direct

$EQ(p, q)$: To check equivalence of FDs p and q

1. Create node (p, q) (if it doesn't already exist),
   and explore it as follows.
2. If it is known that $p = q$ from table of results,
   label node (p, q) 'True', and return.
3. If it is known that $p \neq q$ from table of results,
   label node (p, q) 'False', and return.
4. Check equivalence of Basis BDDs for p and q.
   If not equal, $p \neq q$, label node (p, q) 'False' and return.
5. Check equivalence of LIBDDs for p and q as follows:
   (a) Find pairs of corresponding terminal nodes in the LIBDDs for p and q,
   by recursive traversal through the non-terminal nodes of the LIBDD.
   (b) For each pair of corresponding LIBDD terminal nodes:
   - If the terminal nodes represent Boolean constants,
     label (p, q) 'True/False' depending on their equivalence and return.
   - Otherwise, the terminal nodes represents a pair of FDs s and t.
   - Create a node (s, t) (if it doesn't exist).
   - Create an edge from (p, q) to (s, t).
   - If node (s, t) was just created,
     - Explore it recursively using $EQ(s, t)$.
     - If any node (s, t) is labeled 'False',
       label node (p, q) 'False' and return.

   (c) Reduce graphs of LIBDDs for p and q,
   based on equivalence results of terminal nodes.
6. At this point, $p = q$.
   Label node (p, q) 'True'.

Figure 3.9: Algorithm for Checking LIF Equivalence

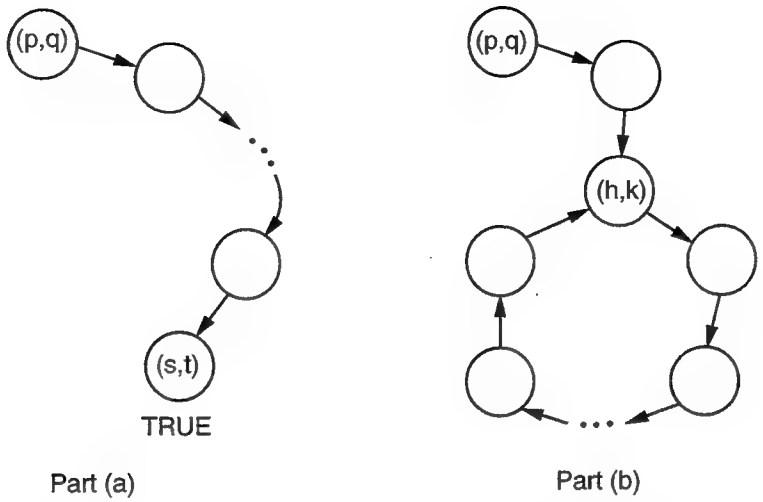


Figure 3.10: Directed Paths in a Comparison-Graph

path from (p, q) to a 'False' node, it is ensured that (p, q) is completely explored, i.e. all its required dependency edges have been constructed. Thus, examining all directed paths from (p, q) ensures that all dependencies are examined.

Each directed path from (p, q) satisfies one of the following conditions:

- it ends in a node labeled 'True', or
- it connects (p, q) to a cycle.

It can now be shown that the dependency chain along a directed path is satisfied under both conditions. Let the node labeled 'True' represent the FD pair (s, t) in the first condition, and let the first cycle node on the directed path represent the FD pair (h, k) in the second condition, as shown in Figure 3.10 (Parts (a) and (b) respectively).

For the first condition, let length of the directed path to the 'True' node be j ($j \geq 1$). Note that the node labeled 'True' indicates that $s^i = t^i$ for $i \geq 1$, known due to result of a previous computation. The path from (p, q) to (s, t) results from the recursive application of Step 5 in the top-level call to $EQ(p, q)$, which checks LIBDDs of the FD pair. Observe that each recursive call of $EQ(p, q)$ *lowers* the implicit parameter of the LIBDDs, since it involves checking the equivalence of FDs pointed to by the terminal nodes, where these pointers implicitly carry the substitution $(i - 1)/i$. Thus, a j-length path lowers the level parameter by j. Using this fact, and the dependency semantics of the Comparison-graph, it can be inferred that the j-length path from (p, q) to (s, t) allows $p^i = q^i$ for all $i > j$. As for $1 \leq i \leq j$, the equivalence $p^i = q^i$ is allowed due to equivalence of the Basis BDDs (checked by Step 4) for all $(j + 1)$ FD pairs along the path (including the pair (p, q)). Thus, this path allows $f^i = g^i$ for all $i \geq 1$.

For the second condition, let length of the non-cycle path be j ($j \geq 0$), and let length of the cycle be n ($n \geq 1$). First consider the cycle node representing the pair (h, k). Step 4 of the algorithm ensures equivalence of the Basis BDDs for h and k, i.e. $h^1 = k^1$. By following the edges around the cycle once, a dependency chain is obtained, such that equivalence of h^i and k^i depends upon equivalence of h^{i-n} and k^{i-n}. For a proof by induction on i, we now have the basis condition ($h^1 = k^1$), as well as the inductive step ($h^i = k^i$) using the induction hypothesis ($h^{i-n} = k^{i-n}$). Thus, by induction on i, this cycle allows $h^i = k^i$ for all $i \geq 1$. Using the node (h, k) in place of a 'True' node, an argument similar to the first condition can now be used to infer that the path from (p, q) to a cycle allows $p^i = q^i$ for all $i \geq 1$.

Thus, all directed paths from (p, q) allow $p^i = q^i$, for $i \geq 1$, thereby establishing the equivalence of FDs corresponding to (p, q).

■

Getting back to Example 7 again, the Comparison-graph for checking the equivalence of g and f is shown in Figure 3.11. Since there is no 'False' node in the Comparison-graph, all nodes (g, f), (h, g), (p, h), (q, p), and (r, q) represent true equalities, i.e. the FDs for all other LIFs (g, h, p, q, r) are equal to the canonical FD for f. Thus, the final canonical FD (CFD) for all LIFs in the system is as shown in Figure 3.12. This completes the task of obtaining canonical LIF representations for this example. In the general case, all FD pairs may need to be examined in order to make them canonical.

3.2.3 Symbolic Manipulation of LIFs

Before moving on to assess the complexity of the algorithms described above, it is useful to consider algorithms for symbolic manipulation of LIFs, since they are closely tied to the method used for obtaining canonical LIF representations.

3.2.3.1 Boolean Operations

As mentioned earlier, the set of LIFs is closed under Boolean operations, i.e. the result h of a Boolean operation on arbitrary LIFs f and g is also an LIF. This is a generalization of the case considered in the previous section where f and g were user-defined LIFs. The difference is that instead of f and g being associated with meta-variables in the meta-space, in the general case, they can be associated with multi-node meta-BDDs. The high-level algorithm for the *Apply* operation on LIFs, called *Lif_Apply*, is summarized from the previous section and is shown in Figure 3.13.

Note that the meta-space is again used to keep track of the new LIFs that may need to be generated in this process. An integrated meta-space is used, both to build the initial set of canonical FDs, and to dynamically add Boolean combination LIFs. In Step 3, it is indicated

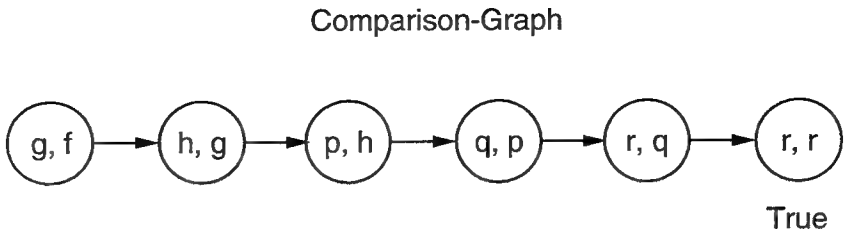


Figure 3.11: Comparison-graph for Example 7

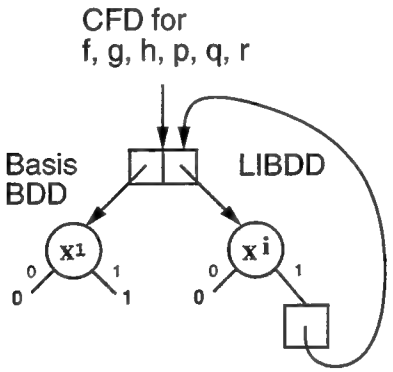


Figure 3.12: Final CFD for Example 7

Lif_Apply: To obtain FD for $h = f < op > g$

1. Obtain the meta-BDD associated with h (meta-BDD(h)), where meta-BDD(h) = meta-BDD(f) $< op >$ meta-BDD(g).
2. If meta-BDD(h) exists in the system already, use the corresponding FD as the FD for h.
3. If meta-BDD(h) does not exist in the system already, generate a new FD for h by using function composition on meta-BDD(h), i.e. compose corresponding FDs for meta-variables in meta-BDD(h).
4. Make the FD for h canonical by comparison against all members in the current canonical set.

Figure 3.13: Algorithm for *Lif_Apply*

that the FD for LIF h is generated by composing in the FDs corresponding to the meta-variables in meta-BDD(h). This view is purely a matter of convenience. It is equivalent to obtaining the FD for h by explicitly performing the $< op >$ operation on the FDs for f and g, as described earlier. In fact, with the latter view, it is easier to see that the total number of new LIFs generated in *Lif_Apply* is bounded by the product of the *transitive closure* sizes for f and g. The *transitive closure* of an LIF f is naturally defined as the set of all LIFs that are accessible from the FD for f, by transitively following all pointers from the LIBDD terminal nodes. Note also that Step 4 can be regarded as a clean-up step, similar to the reduction transformations used locally after each recursive call in the BDD *Apply* operation [18].

As an example which illustrate some new steps, consider the following example:

Example 8: Again, there are three user-defined LIFs e, f, and g, and the Boolean operation $h = f \wedge g$ is required. The user-definitions of e, f, and g are as follows:

for $i = 1, e^1 = x^1 \wedge y^1$ for $i > 1, e^i = (x^i \wedge y^i) \vee ((x^i \oplus y^i) \wedge e^{i-1})$
for $i = 1, f^1 = x^1 \wedge y^1$ for $i > 1, f^i = (x^i \wedge y^i) \vee e^{i-1}$
for $i = 1, g^1 = x^1 \vee y^1$ for $i > 1, g^i = x^i \vee (y^i \wedge g^{i-1})$

Suppose the canonical FDs for e, f and g have already been obtained, as shown in Figure 3.14(a), where the canonical set consists of $\{e, f, g\}$. To obtain the FD for $h = f \wedge g$, meta-BDD(h)=meta-BDD(f) $\wedge$ meta-BDD(g) is first obtained, as shown in Figure 3.14(b). Since it is a distinct meta-BDD, a new FD is generated by composing in the FDs for the corresponding meta-variables e, f, and g in meta-BDD(h). This, in turn, requires a new operation, defined $p^i = e^i \wedge g^i$, and the FD for p is obtained by a similar method. The TFDs for h and p are shown in Figure 3.14(c). For the final step, the TFD for h is made canonical by checking against TFD for e. By using $EQ(h, e)$, the Comparison-graph shown in Figure 3.14(d) is obtained. Since there are no 'False' nodes in the Comparison-graph, equalities denoted by both nodes (h, e) and (p, e) are true. Thus the canonical FDs for both h and p are the same as that for e, and no new member needs to be added to the canonical set.

3.2.3.2 Restriction and Quantification

For LIFs, the Restriction operation can be considered for the following two cases:

- Restriction of an i-instance function f^i, for an i-instance variable $x^i = 0/1$
- Restriction of an LIF f, for all i-instances of a parameterized variable x, i.e. $x^i = 0/1$, $i \geq 1$

In the first case, the operation is exactly like a BDD Restriction operation [22], applied either to the Basis BDD for the case $i = 1$, or to the LIBDD for the case $i > 1$. Note that the

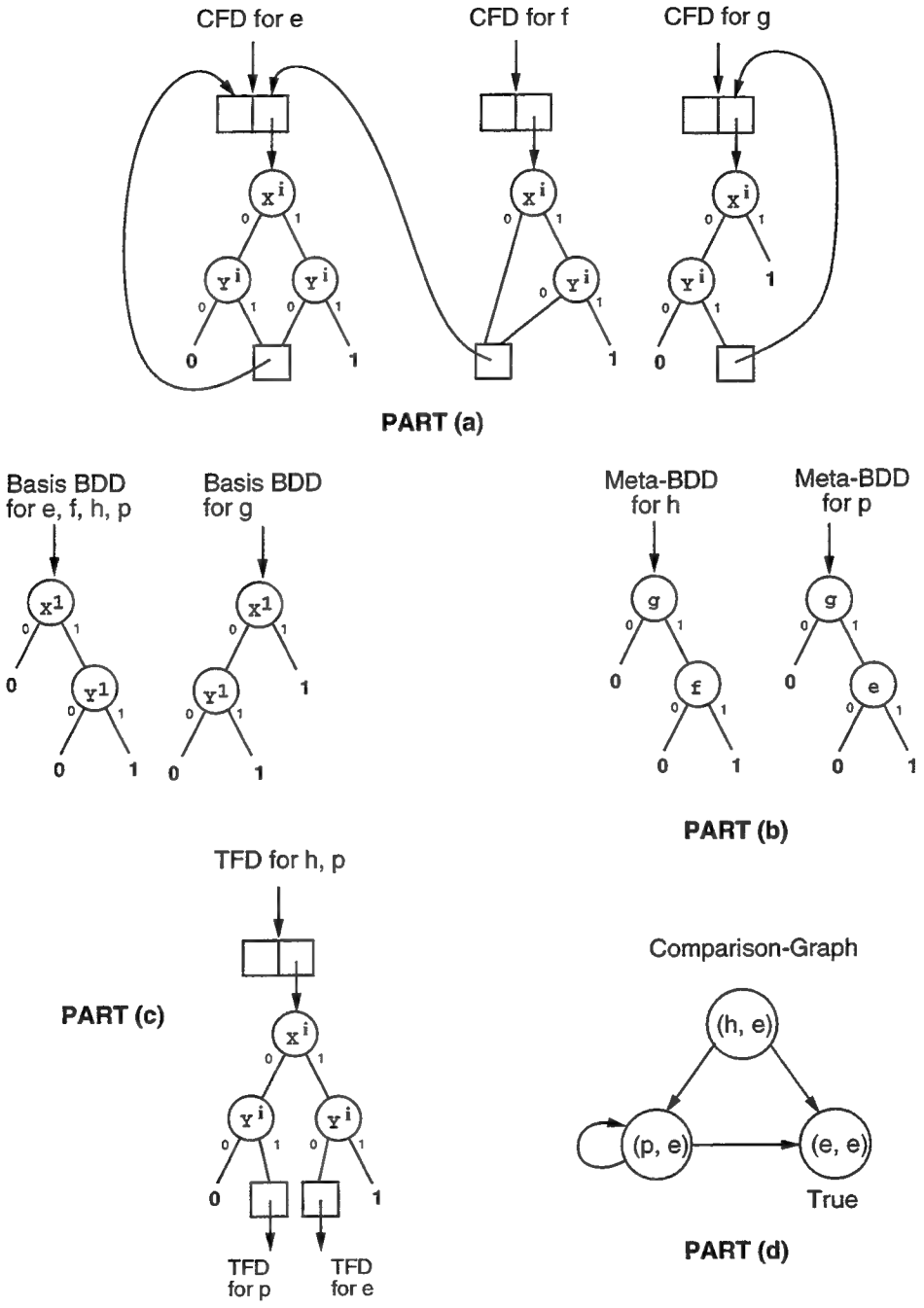


Figure 3.14: Example 8: Boolean Operations for LIFs

terminal nodes of an LIBDD will remain unaffected in such an operation, since they represent $(i-1)$-instances of LIFs, which are independent of x^i.

In the second case, since all instances of a parameterized variable x are restricted to a Boolean constant, the restriction operation has to be performed for both the Basis BDD and the LIBDD. For the Basis BDD, the standard BDD Restriction operation is used [22]. For the LIBDD, the standard operation is modified to account for the LIBDD terminal nodes. Note that these nodes are affected in this case, because the corresponding $(i-1)$-instance LIFs depend upon the restriction for $x^{(i-1)}$. In a manner similar to handling Boolean operations, a new LIF is generated to denote the result of a Restriction operation for a given argument LIF. Again, in order that this generation of new LIFs terminates, all references to a particular restriction of an LIF should be renamed to be the same new LIF. (In an implementation, this can be accomplished easily by assigning a unique operation identifier to each operation, described in detail later.) Since this operation affects only the LIFs in the transitive closure of f, the renaming process is guaranteed to terminate. The follow-up step of making the result FD canonical is the same as in other operations.

Existential and Universal quantification operations can be implemented using the Restriction operation and Boolean operations, as mentioned in Section 3.1.2. Again, two different cases can be defined – quantification with respect to a single instance of a parameterized variable, or quantification with respect to all instances of a parameterized variable. The first case follows exactly the standard BDD operations [22]. For the second, the BDD operation can be modified to handle the LIBDD terminal nodes, in the same way as for the Restriction operation. A new LIF is generated to denote the result of a particular quantification operation on an argument LIF. Suppose a new LIF g is generated to denote the result of a particular quantification operation on f. Note that the required Boolean operations – $\vee$ for Existential quantification, and $\wedge$ for Universal – may involve Boolean combinations of g and other LIFs. These combinations are handled exactly as before, and the FD for g is obtained by using the quantification operation on the FD for f, and is later made canonical in the usual way.

3.2.3.3 Composition

The Composition operation can be viewed in many different ways due to the interaction of parameters allowed by the inductive definitions of LIFs. In this section, only the simple cases are described; the more interesting ones are described in the next chapter, in the context of practical hardware designs.

- Single-instance composition — for a given i, f^i is substituted as an i-instance input x^i in g^i (denoted $g^i|_{x^i=f^i}$). This can be represented by Boolean operations and restriction [22] as:

$$g^i|_{x^i=f^i} = f^i \wedge g^i|_{x^i=1} \vee \neg f^i \wedge g^i|_{x^i=0}$$

These operations are easily handled using the Restriction operation and the Boolean operations described above. Note, again, that the substitution for x^i does not affect any $(i-1)$-instance LIFs appearing in the LIBDD terminal nodes of g^i.

- All-instance composition — for all $i \geq 1$, f^i is substituted as an i-instance input x^i in g^i.

  This case is handled in a manner similar to the all-instance case for the quantification operations. Again, a new LIF is generated to denote the result of a particular composition operation on a given LIF, followed by the required Boolean operations to obtain its FD.

3.2.4 Complexity Analysis for Symbolic LIF Manipulation

As described in the previous sections, the method for obtaining canonical FD representations for LIFs centers around the task of generation and comparison of tentative FDs. The number of tentative FDs is bounded by the number of distinct Boolean combinations of user-defined LIFs in the system. For u number of user-defined LIFs, the maximum number of TFDs is 2^{2^u}. In most cases, the number u is an independent parameter of the problem description. However, the number of variables in the system also places a bound on the maximum number of distinct LIFs that can be defined by a user. In particular, for m basis variables, and n parameterized variables, the number of distinct LIF definitions is $(2^{2^m})^{2^n} = 2^{2^{n+m}}$. This bound is hardly ever reached in practice. Therefore, rest of the complexity analysis is in terms of u, the (independent) number of user-defined LIFs.

The LIF Generation Phase consists of building TFDs for the user-defined LIFs and the newly generated LIFs. For each LIF, this involves first obtaining the meta-BDD, and then using function composition to obtain the corresponding FD (Basis BDD and LIBDD). For meta-BDD manipulations, the complexity is $O(2^{2^u})$. For function compositions, the complexity for Basis BDDs and LIBDDs is $O(2^m \times 2^{2^u})$ and $O(2^n \times 2^{2^u})$, respectively, where m and n are number of basis variables and parameterized variables, respectively.

In the LIF Comparison Phase, the algorithm EQ is used to check equivalence of pairs of FDs recursively. Each top-level application of the algorithm constructs a Comparison-graph, where the number of equivalence results obtained is of the same order as number of nodes in the Comparison-graph (Theorem 2, Section 3.2.2.2). Since a table of results is used across multiple applications of the algorithm, the total number of Comparison-graph nodes is bound by the maximum number of TFD pairs, i.e. $(2^{2^u} \times 2^{2^u}) = 2^{2^{u+1}}$. Each of Steps 1, 2, 3, and 4 of the EQ algorithm requires a constant amount of work per node, and Step 5 requires $O(2^n)$ work per node for the LIBDD traversal. The subsequent reachability analysis of the Comparison-graph is also linear in its size, with $O(2^{2^{u+1}})$ nodes and $O(2^{2^{u+1}} \times 2^n)$ edges (corresponding to 2^n dependency edges per node). Thus, complexity of this phase is $O(2^{2^{u+1}} \times 2^n)$.

In practice, the number of TFDs in the system is much less than 2^{2^u}, since it is rare that all Boolean combinations of user-defined LIFs are required. This affects practical complexity of both phases. Furthermore, equivalence classes of FDs can be implemented using the standard Set-Union-Find data structures [79]. This saves explicit equivalence checks for many pairs of FDs, thereby improving the actual running time. Note also that the analysis considers the worst-case exponential complexity for BDD-like operations in all Boolean subspaces — the meta-BDD space, the Basis BDD space, and the LIBDD space. The practical complexity of these operations depends critically on the nature of the applications, as well as the variable orderings chosen. The advantage of the LIF schema is that whatever efforts are successful in improving the performance of BDD manipulations in general, can be directly applied within this framework.

As for the symbolic manipulation of LIFs described in the previous section, most operations are simple extensions of the corresponding BDD operations on the Basis BDDs and the LIBDDs, followed by canonicity maintenance on the resulting and newly generated LIFs. The contribution due to the BDD-like operation per LIF is typically linear in the size of the Basis BDD and the LIBDD, i.e. $O(2^m)$ and $O(2^n)$ respectively. Furthermore, the number of new LIFs generated in a symbolic operation is bounded by the product of the sizes of the transitive closure of each argument LIF. Thus, the dominant term for symbolic LIF manipulation is due to canonicity maintenance, which is of complexity $O(2^{2^{u+1}} \times 2^n)$, as described earlier.

The important point of this analysis is that complexity of symbolic LIF manipulation, using the FD representations, is *independent of* i, the parameter of inductive description. Instead, it depends on the number of FDs in the system. This is the key strength in using the LIF schema for handling iterative hardware designs – reasoning about such designs is independent of the circuit size parameter. In the next section, this is compared against related techniques used by other researchers to handle iterative hardware.

3.2.5 Related Work

In the context of obtaining upper bounds on the OBDD representation of combinational circuits, Berman characterized a class of multi-output functions which can be defined using a parametric *linear system* of equations [10, 11]. He then showed that the size of OBDD representations for such functions grow linearly with the parameter of description. This class is somewhat more general than the class of LIFs, since it does not require all inductive instances ($i > 1$) to be isomorphic. However, it is precisely this additional constraint that allows the size of FD representations for LIFs to be *independent* of the parameter of description. Furthermore, since the focus of Berman's study was very different, there was no provision for using the OBDD representations to reason about *all* instances of the parametric function, or on utilizing induction. All the same, it is interesting that Berman uses a ripple carry adder as the primary example, which also belongs to the class of LIFs. Finally, Berman's definition of *linearity* is

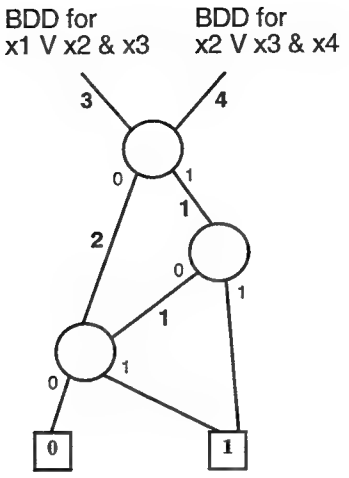


Figure 3.15: Use of BDD Edge Attributes: Variable Shifters

given in terms of OBDD variable orderings, unlike the LIF definition that focuses only on the parametric description. Separating the LIF definition from representation issues, potentially allows easy incorporation of any new advances made in the representation of Boolean functions, including any improvements in the OBDD representation itself.

It is interesting that some elements of the LIF representation have also been studied by other researchers, especially in the context of iterative Boolean functions. In particular, the technique of parameter substitution on LIBDD pointers is similar to an edge attribute proposed by Minato, Ishiura and Yajima [116], called "variable shifters". Basically, when two BDD subgraphs are isomorphic except for a difference in the indices of the input variables, then they can be represented with one copy of the subgraph where this relative difference is stored as the variable shifter along each edge. For example, the functions $(x_1 \vee x_2 \wedge x_3)$ and $(x_2 \vee x_3 \wedge x_4)$ can be represented by the same BDD subgraph, as shown in Figure 3.15, where the variable shift is marked along each edge in bold numbers [116]. However, this technique was used only to obtain greater sharing among BDD subgraphs. No effort was made to exploit it for parameterization in the BDD representation. Naturally, there was also no support for general symbolic manipulation (with canonicity) for iterative functions. Similarly, the concept of "layered" variables was also used by Jain *et al.* [87] to introduce multiple copies of input variables. Again, the focus was on more efficient representations than simple BDDs, without any emphasis on parameterization. It is interesting, all the same, that their general observations regarding decomposition of Boolean functions (for the purpose of generating layers) apply equally well in the context of parametric iterative functions.

On the other hand, a different effort by Ohmura, Yasuura and Tamaru [120] did recognize the potential of parameterization in BDD representations. They described a method for extracting functional information from BDD representations of combinational circuit netlists, by com-

parison against "iterative forms" stored in their database in the form of algebraic recurrence formulas. However, they too did not pursue parameterization of the BDD representation itself.

Finally, the most closely related work is on inductive verification of iterative systems by Rho and Somenzi [130]. They start from the classic descriptions of iterative array systems [74], and use automata techniques to obtain linear-sized OBDD representations for the array outputs. In contrast, the LIF representations are independent of the size parameter, and the technical differences between classic iterative array systems and LIFs have already been summarized in Section 2.1.3, Chapter 2. Furthermore, the focus and details of the respective techniques are also very different. First, their focus is only on inductive verification of such systems. Though their method for verification is independent of the system size, it has not been generalized to provide a framework either for canonical representation, or for symbolic manipulation, of iterative functions. Second, even for the purpose of obtaining linear-sized OBDD representations, they rely on an explicit description of the finite state automaton corresponding to a cell of the iterative array, which considers all combinations of the possible outputs. In contrast, the LIF manipulation method uses only the (implicit) logic equations for each cell and considers outputs individually (without working explicitly with their combinations). The exact details of these two methods will also be compared later in Chapter 5, where the relationship of LIF manipulations and classic automata techniques is described. Third, many aspects of the LIF schema can be generalized to handle the class of EIFs also, as described later in this chapter, thereby placing it in the larger context of handling IBFs in general. It is not clear how the methods used by Rho and Somenzi can be generalized for this purpose. On the other hand, they have extended their approach to other interesting classes of iterative sequential circuits, such as trees and rings [131].

Thus, though there has been prior work on OBDD representations for iterative Boolean functions, those approaches cannot handle symbolic manipulation of all instances of the parametric functions. Furthermore, even for the case of reasoning about a particular i-instance of a parametric function, there may be an advantage in using the LIF manipulation framework, since complexity of the LIF representations and symbolic operations is independent of i. In contrast, complexity of equivalent OBDD manipulations would be typically linear in the size of the corresponding OBDD, where this size is linear in i for iterative circuits.

3.2.6 LIF Implementation Package

In this section, the salient features of a prototype implementation of an LIF package are described. The package is designed as a library, which can be used as the core engine for performing symbolic LIF manipulation within broader application packages.

Conceptually, the LIF package can be viewed as having three different Boolean subspaces:

- Meta-BDD space
  Each meta-BDD provides a handle for the FD associated with an LIF in the system – meta-variables denote user-defined LIFs, and (non-variable) meta-BDDs denote distinct Boolean combinations of user-defined LIFs.

- Basis BDD space
  It consists of Basis BDDs for FDs, defined over basis instances of parameterized variables.

- LIBDD space
  It consists of LIBDDs for FDs, defined over inductive instances of parameterized variables.

Each of these Boolean subspaces is implemented in the style of a standard BDD package, along with the associated memory-management strategies. The prototype is based on the BDD package developed by David Long at CMU [103]. Each Boolean subspace possesses its own unique table, where an entry for a node v denotes the triple $< var(v), hi(v), lo(v) >$. For the three different subspaces, the $var(v)$ field represents the appropriate Boolean variable, and the $hi(v)$ and $lo(v)$ are entries in the same unique table. The only exception is in handling the LIBDD terminal nodes with a pointer to an FD – in this case, the $var(v)$ field contains a pre-set value, and both $hi(v)$ and $lo(v)$ point to the meta-BDD unique table entry associated with the corresponding FD. The variable ordering for each subspace is independently specified by the user.

Each Boolean subspace also contains an operation cache, which supports all standard symbolic Boolean operations. As mentioned earlier, each LIF operation which requires handling the transitive closure of the argument LIF, e.g. restriction, quantification, composition etc., is provided with a unique $operation\_id$. This $operation\_id$ is used to ensure that, within the closure for a particular operation, all references to the result for an argument LIF are bound to the same new LIF. The $operation\_id$ also identifies the variables that are restricted/quantified/composed by associating a variable mapping with an operation.

In addition to the standard BDD-package-style subspaces, the following data structures have been used to support the LIF Generation and Comparison phases:

- A hash table of equivalence results for pairs of FDs – it gets updated each time an equivalence check is performed.

- A Set-Union-Find equivalence class structure on FDs – it is used to union FDs (sets) that have been checked to be equal, and to find the representative FD which is used for subsequent manipulation in place of any FD from the equivalent set.

- A list of canonical FDs – it represents the current canonical set at any point of time, and is initialized to consist of the FD for '1', i.e. an FD where both the Basis BDD and the LIBDD consist of the logical constant '1'.

All standard Boolean operations on LIFs are implemented using appropriate operations in the three Boolean subspaces, followed by canonicity maintenance. Two top-level operations are provided for canonicity maintenance:

- *lif_equal(f,g)* – checks if FD for f is equal to either the FD for g or its negation.
- *make_canonical(f)* – checks equivalence of FD for f against each member in the list of canonical FDs. If the FD for f is equal to any member, the two equivalence classes are merged. If the FD for f is distinct, it is added to the canonical set, and all its LIBDD terminal nodes are recursively made canonical.

Currently, an overall lazy-evaluation approach has been adopted in the LIF package. First, canonicity of FDs is not enforced unless requested by the user. The reason is that unlike BDDs, where canonicity can be syntactically ensured by construction, canonicity for FDs requires additional work. This effort may, or may not, be required in all applications. Therefore, the flexibility currently lies with the user whether or not to enforce canonicity. Second, as described earlier, the FD for an LIF, which denotes a Boolean combination of user-defined LIFs, is obtained by using function composition on the meta-BDD for the LIF. Again, in certain cases, meta-BDD space manipulations may be adequate to ascertain LIF equivalence, without necessarily requiring a check on the corresponding FDs. In fact, it is not even necessary that the LIF Generation phase be complete before starting the LIF Comparison phase. (It was purely for convenience that the description for Example 7 followed this approach.) These phases can be conveniently interleaved, and FDs corresponding to meta-BDDs can be obtained lazily, as and when required.

Practical results from the use of this prototype LIF package are described in Chapters 5 and 6, along with a detailed description of the applications of the IBF methodology.

3.3 EIF Schema

The focus of this section is on the description of the schema for EIFs, with special emphasis on the features common with the LIF schema. The EIF representation uses an Exponentially Inductive BDD (EIBDD) in much the same way as an LIF representation uses an LIBDD. In fact, both the EIBDD and the LIBDD can be viewed as special cases of a generalized Inductive BDD (IBDD) representation[1], also described in this section. The IBDD framework can be potentially useful for representation of other classes of IBFs as well. The algorithms used to obtain canonical EIF representations, and to symbolically manipulate EIFs, are similar to those described for the LIFs. This further indicates potential for a general framework for handling different classes of IBFs in a uniform manner.

[1]The IBDD acronym used here is different from the IBDD used to denote *Indexed* BDDs by other researchers [87].

3.3.1 EIF Representation

An EIF is also identified with an FD, consisting of a tuple of structures:

- Basis BDD – which represents the basis instance of the EIF (for $i = 0$), and
- EIBDD – which parametrically represents all inductive instances of the EIF ($i > 0$).

As in the case of LIFs, the Basis BDD consists simply of a standard BDD over variables that represent the scalar inputs constituting the 0-instance vector input. The ordering on variables is the same as that specified for the corresponding parameterized inputs.

For the EIBDD, first consider a generalized Inductive BDD (IBDD) representation described as follows. Let an IBDD be an extension of the standard BDD, which allows nodes to represent *functions on sets of variables*. Thus, a BDD can be viewed as a special case of an IBDD where each non-terminal node represents a trivial identity function on a singleton set (a single variable). Other extensions with nodes representing Boolean functions have been studied by other researchers – Free Boolean Diagrams [13, 88, 134], Mod-2-BDDs [61] etc. The main objective of these efforts is to provide Boolean functions representations which are more compact than BDDs, especially in the case of some known classes of functions that have exponential-sized BDDs. However, the focus in IBDD representation is primarily on introducing parameterization into the existing BDD framework. Therefore, the details are quite different, as described in the following sections. Furthermore, the techniques for introducing parameterization are in some sense orthogonal to other BDD-related techniques. In principle, they can be combined with whatever is effective in improving the general BDD-based representation.

3.3.1.1 Inductive BDD (IBDD) Representation

Let nodes that represent functions on sets of variables be called *super-nodes*. Define the support set of an super-node to be the set of variables that the corresponding function explicitly depends upon. For example, for a standard BDD node, the support set consist of the singleton variable that it represents. Consider a partition of the complete set of variables in a system of Boolean functions into p (disjoint) sets denoted $S_1, S_2, \ldots S_p$. Define an IBDD to be a BDD-like DAG on super-nodes, i.e. each super-node has two outgoing edges corresponding to the value of the super-node function being a '1' or '0', and terminal nodes denote Boolean constants. The following additional restrictions are also placed:

1. The support set of each super-node in the DAG is a subset of an S_k ($1 \leq k \leq p$).
2. Along any path from the root to a terminal node in the dag, the S_k's corresponding to the super-nodes on that path are disjoint, i.e. no S_k is examined more than once.

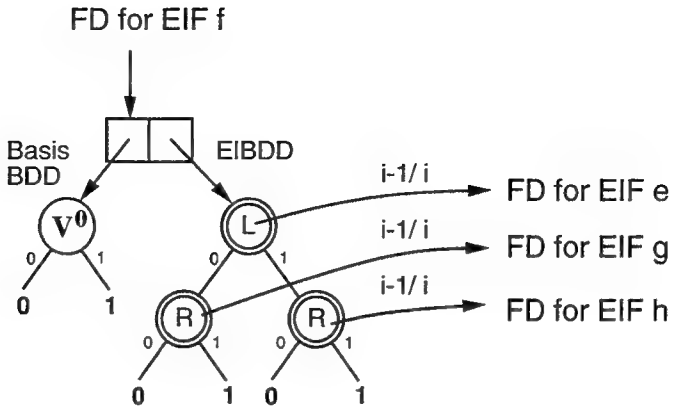


Figure 3.16: A General EIF Representation

The idea is to associate an IBDD with an i-instance of an IBF, and to associate the super-nodes of the IBDD with $(i-1)$-instances of IBFs. This allows the support sets of the super nodes to reflect the explicit pattern of dependence of an i-instance IBF on the parametric inputs.

By following an ordering on the sets of variables S_k $(1 \leq k \leq p)$, Ordered IBDDs are obtained. Again, the standard BDD reduction transformations are extended to regard two super-nodes as equivalent, if they denote the same function and have identical support sets. This results in Reduced Ordered IBDDs. Now, consider a Boolean function f denoting a Boolean combination of functions G on the sets of variables S_k, $1 \leq k \leq p$. It is easily established by the canonicity property of OBDDs, that a Reduced Ordered IBDD is a canonical representation for f, *provided* canonicity of super-nodes representing G can be established. In the IBF framework, since the functions G denote $(i-1)$-instances of IBFs in the system, an inductive argument on i can be used to establish canonicity.

3.3.1.2 Special Cases of IBDDs: EIBDDs and LIBDDs

An EIBDD is a special case of an IBDD with only two partitions on the set of variables – S_L and S_R – denoting the left half and the right half of the input vector, respectively. Typically, a fixed ordering $S_L < S_R$ is used (unless specified otherwise). The EIBDD representation for the inductive i-instance $(i > 0)$ of a general EIF f is shown in Figure 3.16.

In the figure, a super-node is shown as a double-circled node. It contains a pointer to the $(i-1)$-instance of an EIF, along with the label 'L' or 'R' indicating the associated support set. Note that, again, the pointer uses a parameter substitution $(i-1)/i$ (which is implicit in all EIBDD figures that follow). For example, the FD for the tree parity circuit of Example 5 (Chapter 2) is shown in Figure 3.17.

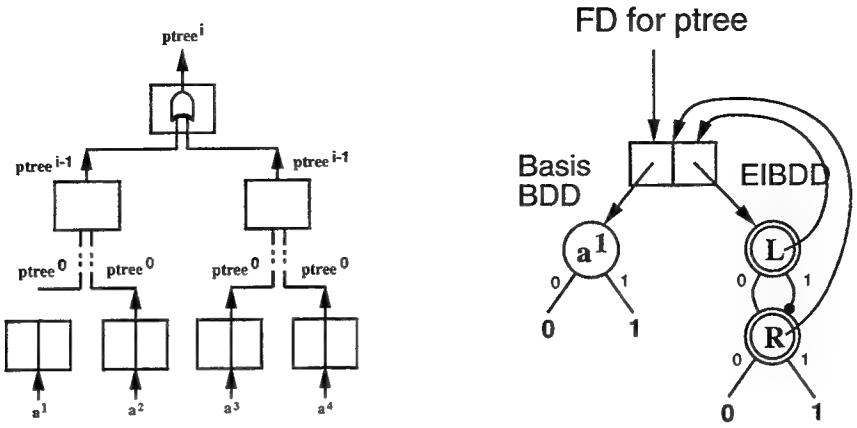


Figure 3.17: EIF Representation for the Tree Parity Circuit

An LIBDD can also be regarded as a special case of the IBDD representation outlined above, where the set of variables is partitioned into:

- $S_1, S_2, \ldots S_n$ — each $S_k, 1 \leq k \leq n$, is a singleton set which denotes an i-instance variable,
- S_{lower} – denotes the set of all remaining lower instance variables.

In this sense, an LIBDD non-terminal node can be regarded as a trivial a super-node (representing the identity function) on a singleton set $S_k, 1 \leq k \leq n$. An LIBDD terminal node with a pointer to an FD, can also be regarded as a super-node representing the corresponding $(i-1)$-instance LIF with support set S_{lower}. Any ordering with $S_k < S_{lower}, 1 \leq k \leq n$, restricts the non-trivial super-nodes in an LIBDD to appear only as terminal nodes.

Note also that this IBDD framework is generalizable for representing different kinds of input partitioning schemes, e.g. dividing the parametric inputs into k sets, computing on them using k inductive subcircuits, and recombining their outputs. Such partitioning forms a crucial component of most inductive hardware designs, and the IBDD representation is geared towards handling this issue explicitly.

3.3.2 Canonicity of EIF Representations

In order to obtain canonical FDs for EIFs, the canonicity of EIBDDs needs to be addressed, since the Basis BDDs can be made canonical by construction. Note from Figure 3.16 that canonicity of each super-node for an EIBDD can be ensured by maintaining canonicity of the

FD that it points to. Once canonicity of super-nodes is established, the remaining EIBDD DAG can be reduced using the standard BDD reduction transformations.

A scheme similar to that for LIFs is used to separate the task into the Generation and Comparison phases. In the Generation phase, tentative FDs (TFDs) are obtained for each user-defined EIF. Note that the user-definitions of EIFs are allowed to express Boolean combinations of other EIFs, *provided* these Boolean combinations are also EIFs. Note that this condition is consistent with Condition 2 of the EIF definition (Definition 2, Chapter 2). This condition is included in the definition itself, because in general, a Boolean combination of two EIFs need not result in an EIF. In other words, unlike the class of LIFs, the class of EIFs is not closed under Boolean operations. However, most practical examples of inductive hardware with recursive tree structure satisfy this condition.

Define an *admissible* Boolean combination of EIFs to be one which results in an EIF. Similar to the case of LIFs, an admissible Boolean combination of user-defined EIFs is renamed to be a new EIF. A meta-space is used to keep track of distinct Boolean combinations of user-defined EIFs. The FD corresponding to a distinct meta-BDD is obtained in the usual way by applying the Boolean operation on the argument FDs. This process is now demonstrated for the carry look-ahead adder example (Example 6, Chapter 2), restated here for convenience.

Example: Consider the definition of $prop$ and gen as follows, representing the carry-propagate and carry-generate functions of a typical log-tree carry lookahead adder, with parametric data inputs $x[2^i]$ and $y[2^i]$.

for $i = 0, prop^0(x[1], y[1]) = x[1] \vee y[1]$
for $i > 0, prop^i(x[2^i], y[2^i]) = prop^{i-1}(x[2^i]_L, y[2^i]_L) \wedge prop^{i-1}(x[2^i]_R, y[2^i]_R)$

for $i = 0, gen^0(x[1], y[1]) = x[1] \wedge y[1]$
for $i > 0, gen^i(x[2^i], y[2^i]) = (gen^{i-1}(x[2^i]_L, y[2^i]_L) \wedge prop^{i-1}(x[2^i]_R, y[2^i]_R)) \vee$
$gen^{i-1}(x[2^i]_R, y[2^i]_R)$

Note that $prop$ is already in the form of an EIF. However, for gen^i ($i > 0$), the definition of $gen^i, i > 0$ is rewritten, as follows:

for $i > 0, gen^i(x[2^i], y[2^i]) = (\neg gen^{i-1}(x[2^i]_L, y[2^i]_L) \wedge gen^{i-1}(x[2^i]_R, y[2^i]_R)) \vee$
$(gen^{i-1}(x[2^i]_L, y[2^i]_L) \wedge (gen^{i-1}(x[2^i]_R, y[2^i]_R) \vee prop^{i-1}(x[2^i]_R, y[2^i]_R)))$

Upon renaming the Boolean combination $temp^i = gen^i \vee prop^i$, gen can now be seen as an EIF:

for $i > 0, gen^i(x[2^i], y[2^i]) = (\neg gen^{i-1}(x[2^i]_L, y[2^i]_L) \wedge gen^{i-1}(x[2^i]_R, y[2^i]_R)) \vee$
$(gen^{i-1}(x[2^i]_L, y[2^i]_L) \wedge temp^{i-1}(x[2^i]_R, y[2^i]_R))$

The TFDs for $prop$ and gen are shown in Figure 3.18(a). The Boolean combination function $temp$ is first represented as a meta-BDD in the meta-space, as shown in Figure 3.18(b). Next, its EIBDD is obtained by using an extension of the BDD *Apply* operation on the EIBDDs for $prop$

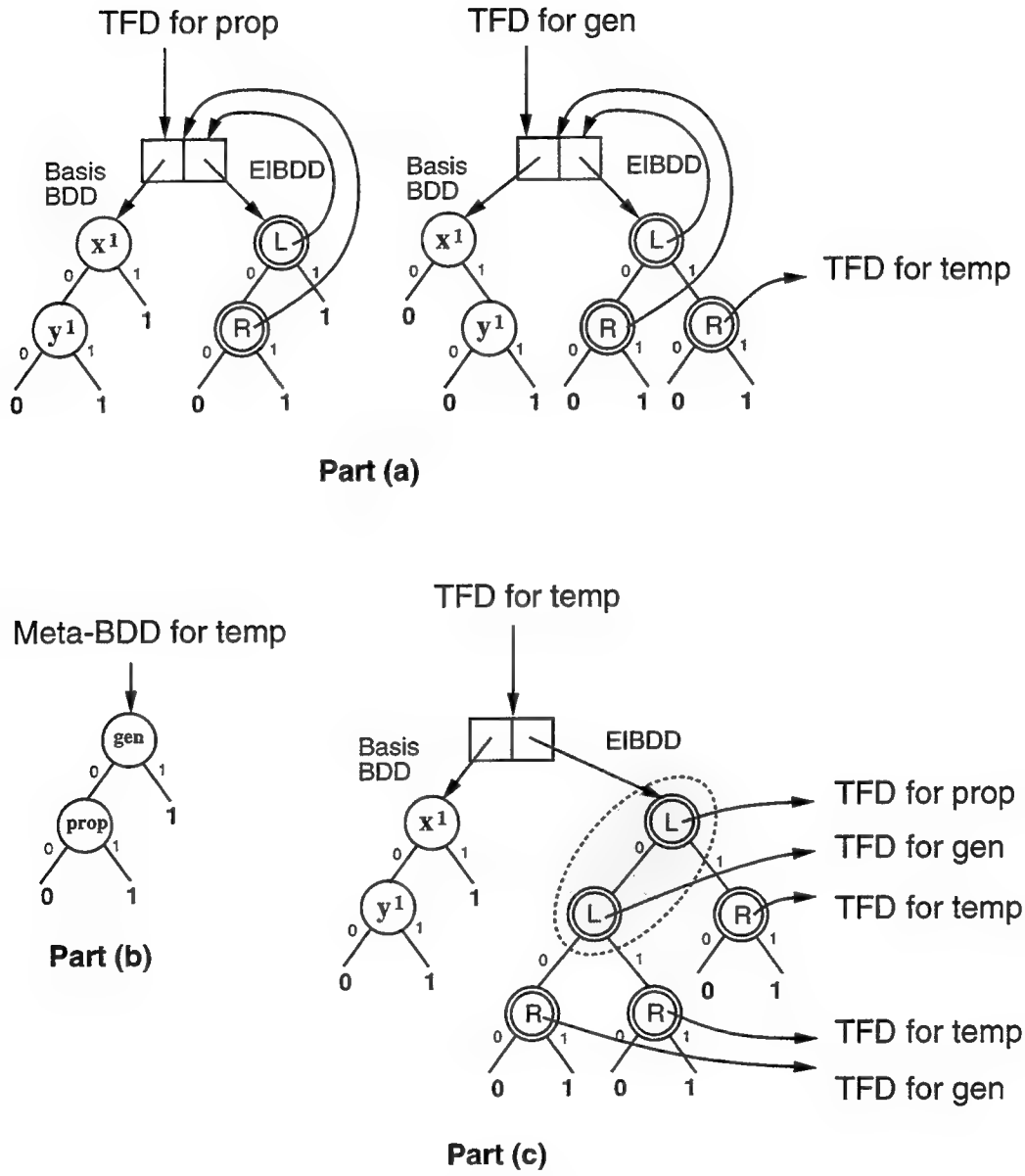


Figure 3.18: TFDs and Meta-Space for Carry Lookahead Adder

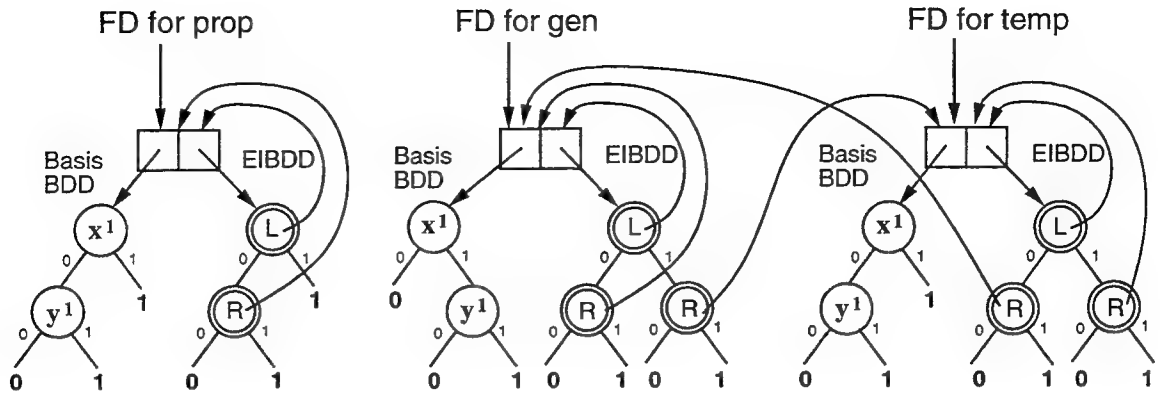


Figure 3.19: Final EIF Representation for Tree Carry Lookahead Adder

and gen. The extension consists of applying the standard operation where each super-node is regarded as being a distinct variable. This is followed by collapsing super-nodes with the same support set into a single super-node, as shown by the shaded region for $temp$ in Figure 3.18(c)). Note that for admissible Boolean combinations of EIFs, Condition 2 of the EIF definition ensures that such collapsing is always possible. In this example, the new function $Temp$ is easily seen to be an EIF.

Once all TFDs in the system are available, they are made canonical in the Comparison phase, by checking for equivalence against members in the canonical set. The test for equivalence uses a modified EQ algorithm (Figure 3.9) along with construction of a Comparison-graph, as described earlier for the LIFs. The simple modification consists of replacing the LIBDD equivalence check (Step 5) with an EIBDD equivalence check, where instead of checking equivalence of corresponding pairs of LIBDD terminal nodes, equivalence of corresponding pairs of super-nodes is checked. Since there are only three super-nodes per EIBDD dag, the Comparison-graph has a simple structure with each node having three outgoing edges. For the example, the TFDs for $prop$, gen, and $temp$ turn out to be canonical, with the final CFDs as shown in Figure 3.19.

3.3.3 Symbolic Manipulation of EIFs

3.3.3.1 Boolean Operations

As described in the previous section, only admissible Boolean combinations of EIFs can be handled in the current framework. To summarize, for an admissible Boolean operation $h = f < op > g$, a tentative FD is generated for h, if it corresponds to a distinct meta-BDD. For this TFD, the Basis BDD is obtained by using the standard BDD *Apply* on the Basis BDDs

for f and g. The tentative EIBDD for h is obtained by using a modified *Apply* (accompanied by super-node collapsing) on the EIBDDs for f and g. This is followed by canonicity maintenance, using a comparison check against each member in the canonical set.

3.3.3.2 Restriction and Composition

Since an i-instance EIF, $i > 0$, does not explicitly depend upon the parametric inputs, the Restriction operation can be defined for following two cases:

- Restriction for a 0-instance function f^0, is defined with respect to a 0-instance variable x^0, and is performed by a standard BDD Restriction operation on the Basis BDD for f.
- Restriction for an i-instance function f^i, $i > 0$, is defined with respect to any of its super-node functions $(e^{i-1}, g^{i-1}, h^{i-1})$ restricted to '0/1'. It is performed by applying the BDD Restriction operation to the EIBDD for f, treating a super-node as a standard node (since the support sets of super-nodes along a path are disjoint).

Composition can be defined in a similar manner, and is obtained by using the Restriction and Boolean operations, as in the case of LIFs.

3.4 Potential for a Generalized IBF Schema

It is clear the LIF schema and the EIF schema are very similar in the algorithms used for obtaining canonical function representations, as well as for symbolic manipulation. The common elements, which can be potentially used for a generalized IBF framework, are:

- An inductive BDD-like representation to capture an i-instance IBF

  This representation can be combined with a standard BDD (which represents the basis instance of the IBF) in order to obtain an FD for representation of all instances of the IBF. The IBDD representation can be viewed as a generalized inductive extension of a BDD, where super-nodes are used to capture lower instances of an IBF. In fact, both the LIBDD and the EIBDD are special cases of the IBDD representation. The main advantage of the IBDD representation is that it allows an inductive argument on the parameter of description, in order to establish canonicity of the representation for an i-instance IBF. Building such an inductive argument into the representation itself, is the precise mechanism which turns out to be very useful for conducting automatic proofs by induction, as described in detail in later chapters. Thus, the IBDD can be potentially used as the foundation of a generalized IBF representation.

- Handling Boolean combinations of IBFs in the Generation Phase

  The use of a separate meta-space to keep track of distinct Boolean combinations of given IBFs is a very important aspect of the IBF schemata. It reflects a sort of "closed-world" assumption about the system of functions under consideration, in that all IBFs in the system are classified in terms of their relationship to a finite user-defined set of IBFs. The assumption that there is such a set, i.e. a set of IBFs for which the user knows the inductive definitions, is a reasonable assumption to make in practice. The payoff is that it allows a conservative estimate of equivalence classes of IBFs, which are subsequently made exact.

- Checking distinctness of FDs in the Comparison Phase

  This is perhaps the weakest aspect of the IBF schemata, albeit necessary in order to obtain final canonical forms. A general exhaustive equivalence check is used to check for distinctness against members of the canonical set. Various optimizations and enhancements to a naive algorithm are possible, such as:

  - Using an equivalence class data structure on FDs to avoid explicit tests for some pairs.
  - Interleaving the Generation phase with the Comparison Phase, such that some equalities may be detected in the meta-space manipulations, thereby avoiding generation of the corresponding FDs.
  - Optimizing on which meta-BDD should be used as a representative for an equivalence class of FDs. This can potentially affect the sizes of the subsequent meta-BDDs.
  - Using random input test sequences in an attempt to distinguish two FDs, before using the exhaustive equivalence check.
  - Using a BDD-based signature matching algorithm to check for equivalence of the first k-instances of two IBFs, before using the exhaustive check on the FDs.

  The top two of these enhancements have been used in the prototype LIF implementation package, and have resulted in performance improvement in practice.

Chapter 4

Parametric Circuit Representation

The focus of this chapter is on representation of parametric circuits using the IBF schemata supplemented with other representation mechanisms for handling general parametrization issues. A brief description of this work can also be found in a preliminary paper [68].

Some examples of practical circuits which can be directly represented as IBFs have already been described in the previous chapter. This chapter starts by formalizing the intuitive notion of parametric inputs for those examples, along with some more interesting examples. Next, the single parameter IBF framework is extended to handle multiple induction parameters. The approach is to concentrate on different kinds of parameter interactions in the definition of a multiple parameter IBF, which are modeled by induction trajectories within a geometric hyperspace framework. The IBF representation is extended to capture these induction trajectories, as well as the different hyperspace regions that characterize a multiple parameter IBF. Again, BDD-style ordering and reduction principles are used to establish canonicity of the extended IBF representations. Though the current framework cannot handle all kinds of parameter interactions, the additional capabilities open up a wide variety of practical applications. These range from handling multi-dimensional structural networks of hardware units to combining the dimensions of space and time for handling inductive ensembles of finite state machines. By introduction of dummy parameters, bidirectional inductive circuits can also be handled in some cases.

Several other parameterization issues arise in dealing with parametric hardware descriptions. These include handling of multiple outputs, vectors of outputs, compositions of representations with different parameters etc. Not all of these can be captured by the extended IBF schemata alone. Therefore, some additional general-purpose representation mechanisms are used. Though this chapter describes their use in conjunction with the IBF schemata, they are not limited to the IBF methodology alone, and can be potentially used with other applications involving parametric hardware descriptions as well.

4.1 Circuit Representation with IBF Schemata: Basics

4.1.1 Representation of Parametric Inputs

The IBF examples described so far in the thesis have used an intuitive notion of what constitutes an *i-instance* of a parametric input. For the LIFs, it is usually regarded as the i^{th}-bit of a vector of inputs. For the EIFs, on the other hand, it denotes the entire 2^i-bit vector, where its two halves provide the implicit $(i-1)$-instance arguments for the $(i-1)$-instance EIFs. In this section, this notion is made precise, which captures the required parameterization for both LIFs and EIFs.

1. A parameterized input x is implicitly represented as an infinite vector of variables $X = [x_1, x_2, \ldots]$.
2. Arguments to IBFs consist of appropriate subvectors of this infinite vector, where a subvector is specified in terms of two components:
   (a) *length* – indicates the number of elements in the subvector
   In the general case, it is represented as an arithmetic expression in terms of the induction parameter. Typically it varies linearly (exponentially) with respect to the induction parameter in an LIF (EIF) schema.
   (b) *offset* – indicates the subscript of the first element of the subvector
   It is also represented as an arithmetic expression in terms of the induction parameter. Again, it usually varies linearly (exponentially) with the induction parameter in an LIF (EIF) schema.
3. A subvector of X with offset k and length n is denoted $X_k[n]$. By convention, the value of k as well as n defaults to '1' (if not explicitly mentioned).
4. Note that the subscript denotes the offset in the infinite vector which is used as a representation for the parametric input, whereas the superscript denotes the parameter of the instance under consideration.

The length/offset notation for input vectors allows expression of parameterized inputs for both the LIFs and the EIFs:

- LIFs — An i-instance LIF with input $X[i]$ depends explicitly upon the i-instance input X_i, and the remaining inputs $X[i-1]$ are implicit arguments of the $(i-1)$-instance LIFs.
- EIFs — For an i-instance EIF with input $X[2^i]$, the left half $X[2^{i-1}]$ and the right half $X_{2^{i-1}+1}[2^{i-1}]$ of the input are implicit arguments of the $(i-1)$-instance EIFs.

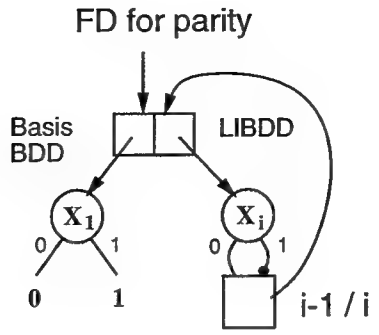


Figure 4.1: LIF Representation for a Serial Parity Circuit

Recall that the set of variables can contain both parametric as well as non-parametric variables. As briefly alluded to in Chapter 2, the latter are allowed without any restrictions in the basis-instance definition of an IBF. However, some restrictions have to be placed on their use in an inductive-instance definition, which are described later in this chapter. From the application point of view, some variables denote control inputs, while others denote data inputs. As a matter of convention for the variable ordering, non-parametric control variables are placed first, then the parametric control variables, followed by parametric data variables. This allows use of non-parametric control variables to steer the choice of an appropriate inductive function, as illustrated later in this chapter.

4.1.2 Structural Induction Examples

Direct representation with IBF schemata is very useful for structurally-inductive circuits where an i-instance circuit has a fixed number of outputs, each of which can be represented by an i-instance IBF. Examples of such descriptions have already been provided in Chapter 2 for an i-bit ripple carry adder (Example 1), an i-bit serial parity circuit (Example 2), and an i-bit comparator circuit (Example 3). The LIF representation for the ripple carry adder has been shown in Figure 3.4, Chapter 3. For the serial parity circuit, and the comparator circuit, it is shown here in Figures 4.1 and 4.2, respectively.

Another interesting example which fully demonstrates the parametric input notation is a multiplexor circuit. Its parametric description can be given as follows:

Example 9: An i-instance multiplexor circuit has two parametric inputs – $X[2^i]$ denoting the 2^i-bit data input, and $S[i]$ denoting the i-bit selector input. The output of the i-instance circuit mux^i, selects the data bit $X_{||S[i]||}$, where $||S[i]||$ denotes the integer value of the binary number $S[i]$. Because of the naturally inductive representation for $||S[i]||$, mux^i can be easily captured as an i-instance LIF:

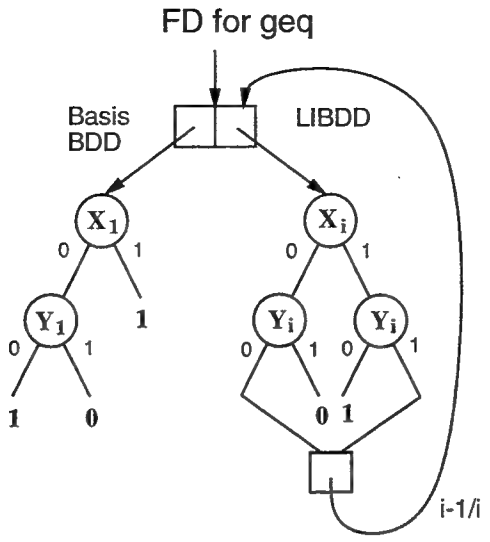


Figure 4.2: LIF Representation for a Comparator Circuit

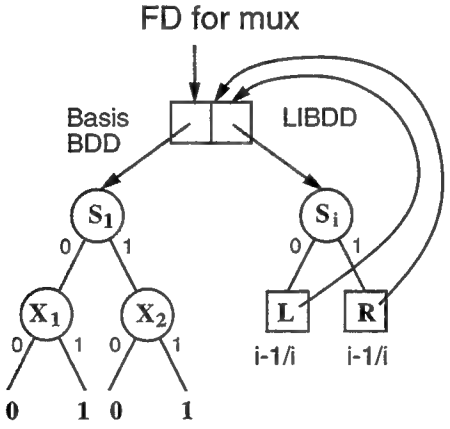


Figure 4.3: LIF Representation for a Multiplexor Circuit

for $i = 1, mux^1 = (\neg S_1 \wedge X_1) \vee (S_1 \wedge X_2)$
for $i > 1, mux^i = (\neg S_i \wedge mux^{i-1}(S[i-1], X[2^{i-1}])) \vee (S_i \wedge mux^{i-1}(S[i-1], X_{2^{i-1}+1}[2^{i-1}]))$

Note that while mux^i explicitly uses the i^{th}-bit of the selector input S, it uses the two halves of the data input X implicitly as arguments to mux^{i-1}. Thus, this example combines elements from both the LIF and the EIF pattern of input dependence. However, since the i-instance input S_i provides an anchor for the explicit relationship between the i-instance LIF and the i-instance input, it can be represented as an LIF as shown in Figure 4.3.

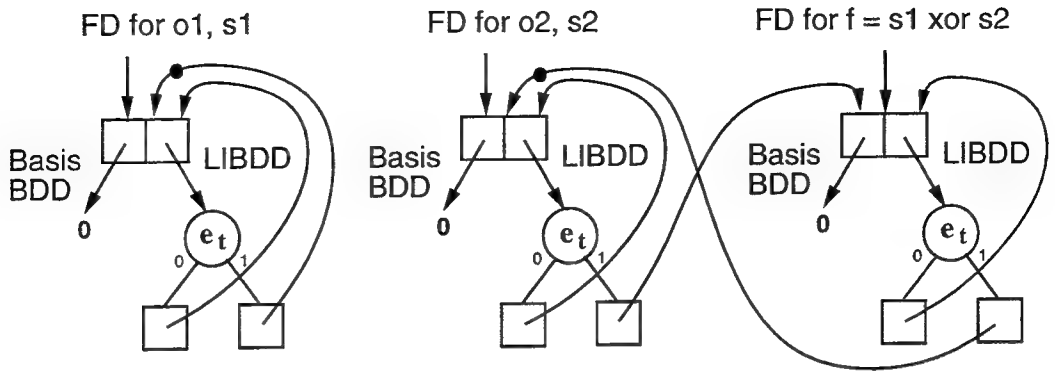


Figure 4.4: LIF Representation for a 2-bit Counter

For EIF examples, the EIF representations for some examples have already been provided – tree parity circuit (Figure 3.17, Chapter 3), carry-propagate and carry-generate functions of a carry lookahead adder (Figure 3.18, Chapter 3). Additional representation mechanisms are needed to capture other binary tree circuits, as described later in this chapter.

4.1.3 Temporal Induction Examples

As described in Chapter 2, a Mealy machine model description for any finite state machine can be directly represented as a system of LIFs, where each state transition function and each output function is individually represented as an LIF. The LIF representation for a standard 2-bit counter (Example 4, Chapter 2) is shown in Figure 4.4. Again, the dark circles on pointers from LIBDD nodes to FDs are negative edge attributes that denote complementation of the functions pointed to. Note that an additional LIF f is used to represent the required Boolean combination function $f = s_1 \oplus s_2$. Note also that the output functions $(o1, o2)$ in this example are identical to the state transition functions $(s1, s2$, respectively). However, this is an artifact of the particular state encoding used. As shown later, the output LIF representations remain the same even when a different state encoding is used, as is expected by the canonicity argument for the LIF representation.

As another example, consider the following behavioral description of a shift register.

Example 10: Consider a parallel-load shift register with a fixed number of cells, say 3, with cell outputs $s1$, $s2$, and $s3$, as shown in Figure 4.5. Each cell has a parallel-load data input – $x1$, $x2$, and $x3$, respectively. Also, there are data inputs at both ends of the shift register – yl at the left, and yr at the right. The two control inputs (p, q) define the operations – $(0,0) : noop, (0,1) : load, (1,0) : shift\_left, (1,1) : shift\_right$. The behavioral description

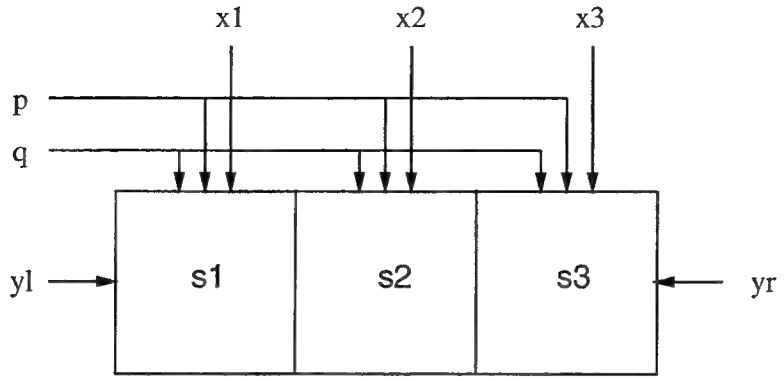


Figure 4.5: A Shift Register Circuit

for each cell output can be captured as an LIF as follows (where the initial output at $t = 1$ is assumed to be '0'):

for $t = 1, s1^1 = 0$,
for $t > 1, s1^t = (\neg p^t \wedge \neg q^t) \wedge s1^{t-1} \vee (\neg p^t \wedge q^t) \wedge x1^t \vee (p \wedge \neg q^t) \wedge s2^{t-1} \vee (p \wedge q^t) \wedge yl^t$

for $t = 1, s2^1 = 0$,
for $t > 1, s2^t = (\neg p^t \wedge \neg q^t) \wedge s2^{t-1} \vee (\neg p^t \wedge q^t) \wedge x2^t \vee (p \wedge \neg q^t) \wedge s3^{t-1} \vee (p \wedge q^t) \wedge s1^t$

for $t = 1, s3^1 = 0$,
for $t > 1, s3^t = (\neg p^t \wedge \neg q^t) \wedge s3^{t-1} \vee (\neg p^t \wedge q^t) \wedge x3^t \vee (p \wedge \neg q^t) \wedge yr^{t-1} \vee (p \wedge q^t) \wedge s2^t$

The LIF representations for these shift register cell outputs are shown in Figure 4.6. Note that the control input variables p^t and q^t are used to steer the choice of the appropriate data value for each shift register cell.

4.2 Multiple Parameter IBF Schemata

In this section, the IBF schemata described in the previous chapter are extended in order to handle multiple parameter IBFs. The idea is to use a geometric hyperspace framework to represent the parameter space of an IBF definition, and use it in conjunction with the FD representation. This framework can be potentially used with other schema for canonical representation of a single parameter function also.

4.2.1 Geometric Hyperspace Framework

A geometric hyperspace is used to represent the parameter space, where each axis corresponds to an induction parameter. Thus, an IBF definition corresponds to following an induction

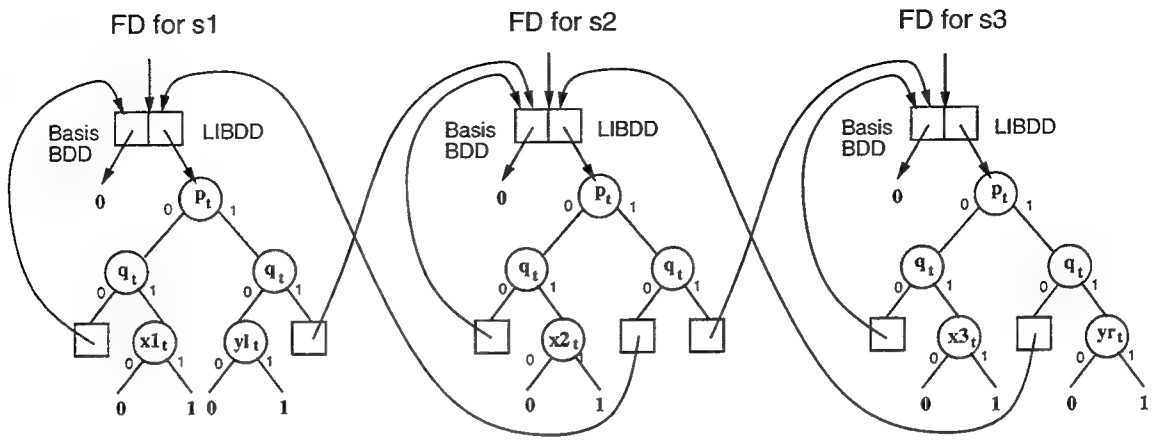


Figure 4.6: LIF Representation for a Shift Register

trajectory in the hyperspace, which describes how the function changes inductively along each step in the trajectory.

Example 11: Consider the following three descriptions for a function f, defined in terms of two induction parameters i and j, where the (i,j)-instance of f is denoted $f^{(i,j)}$:

- **Description 1**: basis case for $i = 1, j = 1$: $f^{(1,1)} = \mathcal{B}_1(X^{(1,1)})$
  inductive case for $i = 1, j > 1$: $f^{(1,j)} = \mathcal{B}_{1j}(X^{(1,j)}, G^{(1,j-1)})$
  inductive case for $i > 1, j > 1$: $f^{(i,j)} = \mathcal{B}_i(X^{(i,j)}, G^{(i-1,j)})$

  With this description, $f^{(m,n)}$ can be obtained by following the solid-line trajectory in the parameter hyperspace as shown in Figure 4.7. Note that it corresponds to first applying induction along j, starting from the basis case $f^{(1,1)}$, and leading to the intermediate function $f^{(1,n)}$. This intermediate function then forms the basis for an induction along i, resulting finally in $f^{(m,n)}$.

- **Description 2**: basis case for $i = 1, j = 1$: $f^{(1,1)} = \mathcal{B}_1(X^{(1,1)})$
  inductive case for $i > 1, j > 1$: $f^{(i,j)} = \mathcal{B}_j(X^{(i,j)}, G^{(i,j-1)})$
  inductive case for $i > 1, j > 1$: $f^{(i,j)} = \mathcal{B}_i(X^{(i,j)}, G^{(i-1,j)})$

  With this description, $f^{(m,n)}$ can be obtained by following any orthogonal trajectory, such as that shown by a dotted line in Figure 4.7. Note that an orthogonal trajectory allows only one parameter to change at a time. This characteristic is also evident in the inductive case definitions for $i > 1, j > 1$.

- **Description 3**: basis case for $i = 1, j = 1$: $f^{(1,1)} = \mathcal{B}_1(X^{(1,1)})$

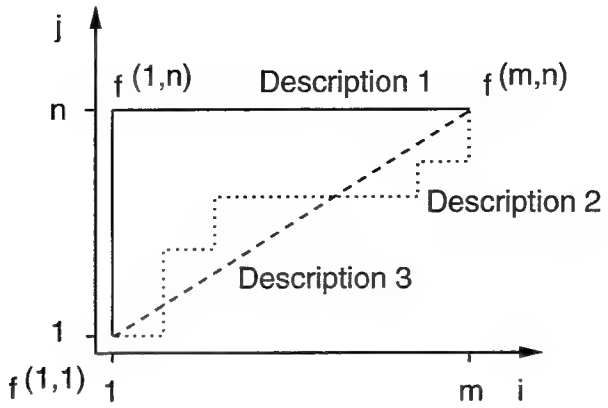


Figure 4.7: Induction Trajectories in Multiple Parameter Hyperspace

$$\begin{array}{ll} \text{inductive case for } i > 1, j > 1 & : f^{(i,j)} = \mathcal{B}_j(X^{(i,j)}, G^{(i,j-1)}) \\ \text{inductive case for } i > 1, j > 1 & : f^{(i,j)} = \mathcal{B}_i(X^{(i,j)}, G^{(i-1,j)}) \\ \text{inductive case for } i > 1, j > 1 & : f^{(i,j)} = \mathcal{B}_{ij}(X^{(i,j)}, G^{(i-1,j-1)}) \end{array}$$

With this description, $f^{(m,n)}$ can be obtained by following any arbitrary linear trajectory, such as that shown by a dashed line in Figure 4.7. Note that in contrast to the previous descriptions, the last inductive case definition in this description allows both parameters to change simultaneously. (Naturally, such a description does permit conflicts among the inductive definitions, and care must be exercised for its proper use.)

The current IBF schemata extensions admit orthogonal trajectories only, i.e. only those inductive function definitions are allowed where parameters change *one at a time*. This is not a serious limitation, since most practical parametric circuit descriptions use independent parameters, whereby induction for one parameter takes place independently of induction for the other. For example, a register file can be described parametrically in terms of two independent parameters – i denoting the number of address lines, and j denoting the number of data lines. In such cases, the definitions correspond to orthogonal trajectories. This limitation is motivated by the simplicity of the accompanying canonicity argument, as described in detail in the next section. Some suggestions are also offered for its potential removal towards the end of this section. However, each of those entails the cost of significant additional work.

4.2.2 Parameter Decision Tree Representation

Recall that the FD representation used a fixed tuple of two structures – one for the basis case, and the other for the inductive case description. This representation can be generalized, where instead of two fixed slots, a binary decision tree on parameters is used. This is called

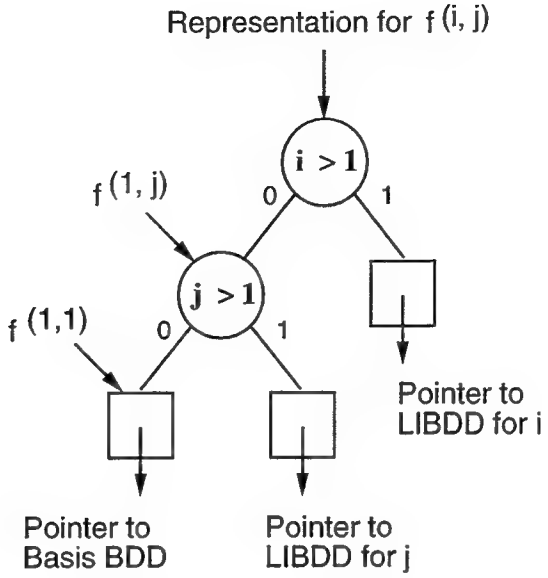


Figure 4.8: Parameter Decision Tree Representation

a *parameter decision tree representation.* An example of this representation is shown in Figure 4.8 for Description 1 of Example 11.

The terminal nodes of a parameter decision tree (indicated as boxes) contain pointers to canonical representations of the basis case and the inductive case descriptions of the function. In the IBF framework, these correspond to the Basis BDDs and LIBDDs (or EIBDDs)[1], respectively. Each non-terminal node of a parameter decision tree (indicated as circles) denotes a comparison of a induction parameter against its basis value (1/0 for LIFs/EIFs, respectively). It has two outgoing edges, corresponding to the comparison test being '1' (true) or '0' (false).

The structure of a parameter decision tree captures an induction trajectory as follows. Consider the non-terminal node for '$j > 1$' in Figure 4.8. Its 0-edge leads to a terminal node with a pointer to the Basis BDD, which represents the definition of f for $i = 1, j = 1$. Its 1-edge leads to a terminal node with a pointer to an LIBDD, which represents the definition of f for an inductive step in j. Therefore, the non-terminal node for j effectively captures the induction along j, where its 0-edge provides the basis case representation, and its 1-edge the inductive case representation. In terms of the induction trajectory (shown as a solid line in Figure 4.7), this node represents the vertical part associated with the function $f^{(1,j)}, j \geq 1$. Now consider the non-terminal node for '$i > 1$'. Its 0-edge, which leads to the non-terminal node for j representing $f^{(1,j)}, j \geq 1$, provides the basis for the induction along i. Its 1-edge leads to an LIBDD representation of the definition of f for an inductive step in i. Thus, this node captures

[1]The LIBDDs and EIBDDs cannot be mixed within a single parameter decision tree.

an induction along i corresponding to the horizontal part of the induction trajectory, and it can be associated with the function $f^{(i,j)}, i \geq 1, j \geq 1$. In this way, *the structure of the parameter decision tree provides the appropriate basis functions for different induction steps along an induction trajectory*. This property is crucial in generalizing this representation for capturing arbitrary induction trajectories.

4.2.3 Canonicity of Multiple Parameter IBF Representations

The canonicity of terminal nodes of the parameter decision tree can be ensured by using the single parameter IBF schemata already developed. In fact, any other schema for canonical representation of a single parameter inductive function would also suffice. Within the IBF schemata, the Basis BDDs and the LIBDDs are now allowed to use variables parametric in any subset of induction parameters. The requirement for canonicity is only that *each parametric instance of a variable should be used only once* in the representation of any function, where the instances are interpreted with respect to the parameter substitution mechanism described earlier. As long as this requirement is satisfied, the LIBDD canonicity argument given earlier still holds.

For rest of the parameter decision tree, a total ordering on the non-terminal nodes is imposed. This ordering, denoted π, is specified as a fixed, user-given ordering on the induction parameters. For example, the parameter decision tree of Figure 4.8 is consistent with $i < j$, but is in conflict with $j < i$ (and would therefore be unacceptable). In a manner similar to OBDDs, an ordered parameter decision tree can be reduced by elimination of redundant nodes and isomorphic subgraphs in order to obtain a canonical DAG[2]. Note that the canonicity of this extended IBF representation holds with respect to the ordering on the induction parameters (used by the parameter decision tree) as well as the ordering on parameterized variables (used by the Basis BDDs and the LIBDDs/EIBDDs).

In this regard, it is interesting to recapitulate the interplay of various ordering constraints used by the multiple parameter IBF representation. This also clarifies their relationship to a global ordering (denoted $<_g$ on all instances of the parameterized variables $X_{(i,j)}$ which could, in principle, be used by an OBDD representation of $f^{(i,j)}$.

- Recall that the Basis BDDs and the LIBDDs use an ordering ρ on different parameterized variables (not their instances). For example, a variable ordering $v <_\rho w$ specifies that $v_{(i,j)} <_g w_{(i,j)}$ for $i \geq 1, j \geq 1$.
- The LIBDD representation assumes that all variable instances associated with the same parameter values are lumped together in a *layer*. Under this assumption, ρ can be viewed as specifying the *intra-layer* variable ordering for all layers.

[2]Technically, the resulting structure may be a DAG; however the term "tree" has been used for convenience.

- Furthermore, the LIBDD representation assumes that the *inter-layer* ordering is such that the layers are stacked in decreasing order of value with respect to each parameter. For example, for each variable v, and for each induction parameter i, the variable instances are ordered such that $v_{(i+1,j)} <_g v_{(i,j)}$, for $i > 1, j \geq 1$. (Similarly $v_{(i,j+1)} <_g v_{(i,j)}$ for $i \geq 1, j > 1$.) However, this still does not place any constraint on layers with different values for different induction parameters.

- Finally, an ordering π on induction parameters is used by the parameter decision tree to further constrain the inter-layer ordering. In effect, π is used to provide ordered tuples of parameters with a lexicographic ordering on all values, such that for $i <_\pi j$,

  $(i_1, j_1) <_g (i_2, j_2) \equiv (i_1 > i_2) \vee ((i_1 = i_2 \wedge j_1 > j_2))$

  where $>$ denotes the standard arithmetic (numerical) greater-than relationship on the integer values. This ordering on tuples of parameters is used as the inter-layer ordering on parametric instances of variables.

Note that a combination of these constraints fully specifies a total ordering $<_g$ on all instances of the parameterized variables. Given the correspondence of the LIBDD/EIBDD representations to OBDDs (Chapter 3), it is clear that the multiple parameter IBF representation is canonical only with respect to π (the ordering on induction parameters), and ρ (the ordering on parameterized variables).

4.2.4 Current Limitations and Potential Extensions

The above discussion also provides some insights into the current limitations of the multiple parameter IBF representation. First, note that the non-terminal nodes of the parameter decision tree allow comparison of induction parameters against their basis values only. In the geometric hyperspace, this is equivalent to forcing induction along one parameter *before* starting induction along another. Thus, it cannot represent an interleaving of induction steps along different parameters even for an orthogonal trajectory, e.g. the dashed-line trajectory in Figure 4.7. The crucial requirement, as remarked earlier, is to capture the appropriate basis functions for the different steps along the induction trajectory. One way to accomplish it is by allowing comparison of induction parameters against arithmetic expressions which characterize the trajectory. However, this complicates the issue of ordering non-terminal nodes based solely on a user-given ordering on the parameters. A similar issue is described later in Section 4.4.1, along with a preliminary approach to handle it.

Another issue is the representation of *simultaneous induction* along multiple parameters, such as the dotted-line induction trajectory in Figure 4.7. The situation is schematically illustrated in Figure 4.9, which shows different kinds of inductive steps in a geometric hyperspace defined by two parameters i and j. Each step is labeled by the underlying variables explicitly used in

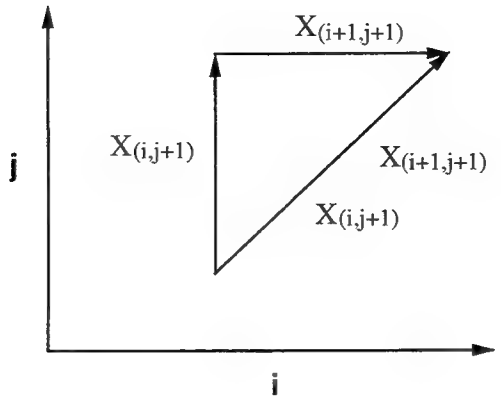


Figure 4.9: Inductive Steps and Underlying Variables in Parameter Hyperspace

that step. Note that the orthogonal steps use either $X_{(i,j+1)}$ or $X_{(i+1,j+1)}$; while a simultaneous induction step uses both $X_{(i,j+1)}$ and $X_{(i+1,j+1)}$.

The main advantage of allowing only orthogonal induction steps is that it directly corresponds to a linear ordering on the underlying *layers* of variables, where each layer lumps together all variable instances with same parameter values. As explained in detail for the LIBDD representation (Section 3.2.1.4, Chapter 3), this allows a simple parameter substitution mechanism on the variables in different layers. Furthermore, it allows the same mechanism for handling a Boolean combination of such functions also. Recall that a new function is generated to denote a Boolean combination of functions with *uniform* parameters, i.e. the same parametric instances. Since the underlying variables (layers) for functions with uniform parameter values are the same, it is possible to substitute for the functions in terms of the variables, in order to obtain a parametric description for the new function itself. Moreover, any new Boolean combinations that arise in this process also have uniform parameters. This property is crucial in handling Boolean combinations by using a simple renaming mechanism, as described in detail in Chapter 3 for both LIFs and EIFs. This renaming mechanism forms an essential component of maintaining canonicity of the IBF representations.

On the other hand, if simultaneous induction is allowed, then each inductive step has to handle multiple layers of variables at a time. This itself may not be a problem for the parameter substitution mechanism on variable instances. However, it may pose a problem for handling Boolean combinations of such functions. Due to different variable layers being handled simultaneously, Boolean combinations of functions with *non-uniform* parameters, i.e. different parameter instances, may be required. For example, a Boolean combination $f^{(i,j)} \vee g^{(i,j+1)}$ may arise due to layers $X_{(i,j+1)}$ and $X_{(i+1,j+1)}$. It is not always possible to represent such a Boolean combination as another IBF, and therefore the renaming mechanism may not work. In some cases, it may be possible to define dummy functions to capture such combinations. However,

in these cases, the underlying effect is still a separation of the different variable layers, which amounts to orthogonal inductive steps.

Therefore, currently, no schema has been provided to handle simultaneous induction in the general case. For those cases which are reducible to orthogonal steps by defining a constant number of dummy functions, a canonical IBF representation can be obtained. For some others, it is possible to provide limited types of symbolic manipulations, while maintaining canonical representations in one parameter only. Practical examples for these cases are described in the remainder of this chapter, as well as in Chapter 6 along with details of the verification applications.

4.3 Combining Parameterization in Space and Time

An interesting class of circuits that can be handled with the multiple parameter IBF schemata are those that involve induction in both space and time. Such circuits occur frequently in practice, e.g. in protocol applications where correct operation of the protocols is required regardless of the number of processes etc. Since the current framework for handling multiple parameter IBFs is not fully general, several different techniques have been explored to tackle the problems in practice. The most straightforward approach, where possible, is of using space and time as independent parameters. This allows the full benefit of canonical IBF representation, along with general symbolic manipulation. In those cases where it is not possible to use independent parameters, for example where the orthogonality constraint may be violated, there are other techniques that may help with symbolically manipulating such circuits. These include introduction of dummy parameters, and introduction of dummy functions. Even though they do not afford a canonical representation, yet they are useful from the point of view of verification applications. The techniques are described, along with illustrative examples, in this section; the verification applications are considered in more detail in Chapter 6.

4.3.1 Independent Parameter Framework

In the case of space and time representing independent parameters, the IBF schemata can be directly used to obtain the canonical representation for the inductive circuit. Datapath circuits where each bit functionality is independent of the others fall in this category. Typical examples include w-bit wide datapaths used for data movement, such as shift registers, stacks, FIFO's etc. Consider a space-parameterized version of the shift register described in Example 10:

Example 12: Let the shift register be parametric in the bit-width also, denoted by parameter i. Note that i denotes the width, not the length of the shift register, as shown schematically in Figure 4.10. In other words, the number of shift register cells in the direction of data

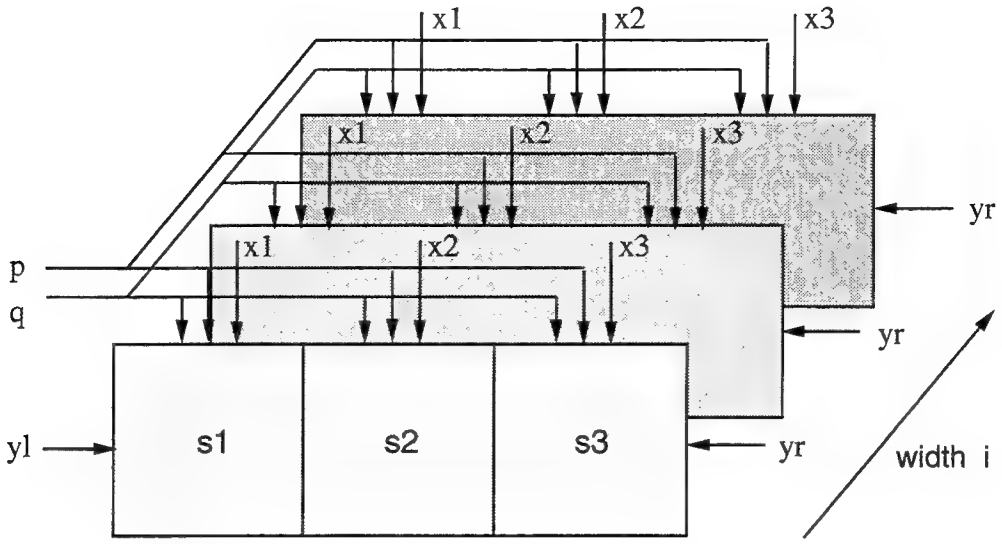


Figure 4.10: Shift Register with Parametric Width

movement is constant (3 in this example), while the number of linear cell arrays operating in parallel is i. As shown in the figure, each cell array i has its own set of data inputs – $x1_{(i,t)}, x2_{(i,t)}, x3_{(i,t)}, yl_{(i,t)}, yr_{(i,t)}$. However, the control inputs, p_t and q_t, are shared by all cell arrays. Note that since the data bits in cell array i are completely independent of data bits in cell array j, $i \neq j$, it is possible to use multiple parameter LIF representations for the cell array outputs. These are shown in Figure 4.11, where $s1^{(i,t)}, s2^{(i,t)}$, and $s3^{(i,t)}$ denote the outputs from the first, second and third cells of the i^{th} array, respectively.

The top part of the figure denotes the parameter space, with simple parameter decision trees with parameter ordering $t < i$. The terminal nodes of the parameter decision trees point to appropriate Basis BDDs and LIBDDs in the variable space, shown in the bottom half of the figure. (The associated meta-BDDs have been omitted.) Note that pointers from LIBDD terminal nodes to FDs again carry an implicit parameter substitution for the particular parameter marked.

Note also that an (i,t)-instance LIF (e.g. $s1^{(i,t)}, s2^{(i,t)}, s3^{(i,t)}$) depends upon a t-instance variable (e.g. p_t, q_t). This is equivalent to an inductive instance of an LIF (with parameter i) depending upon a non-parameterized variable (independent of i). This example illustrates the conditions under which such use is allowed. Note first that the independence between different i-arrays ensures that there is no reuse of p_t and q_t between cells of different arrays. Thus, the only reuse of these variables is between different cells of the same array. In these cases, the parameter substitution $(t-1)/t$ ensures that the representation refers to different parametric instances of p_t and q_t. Thus, the requirement is similar to the read-once-only requirement of BDDs, i.e. *no non-parametric variable, or a particular parametric instance of a variable,*

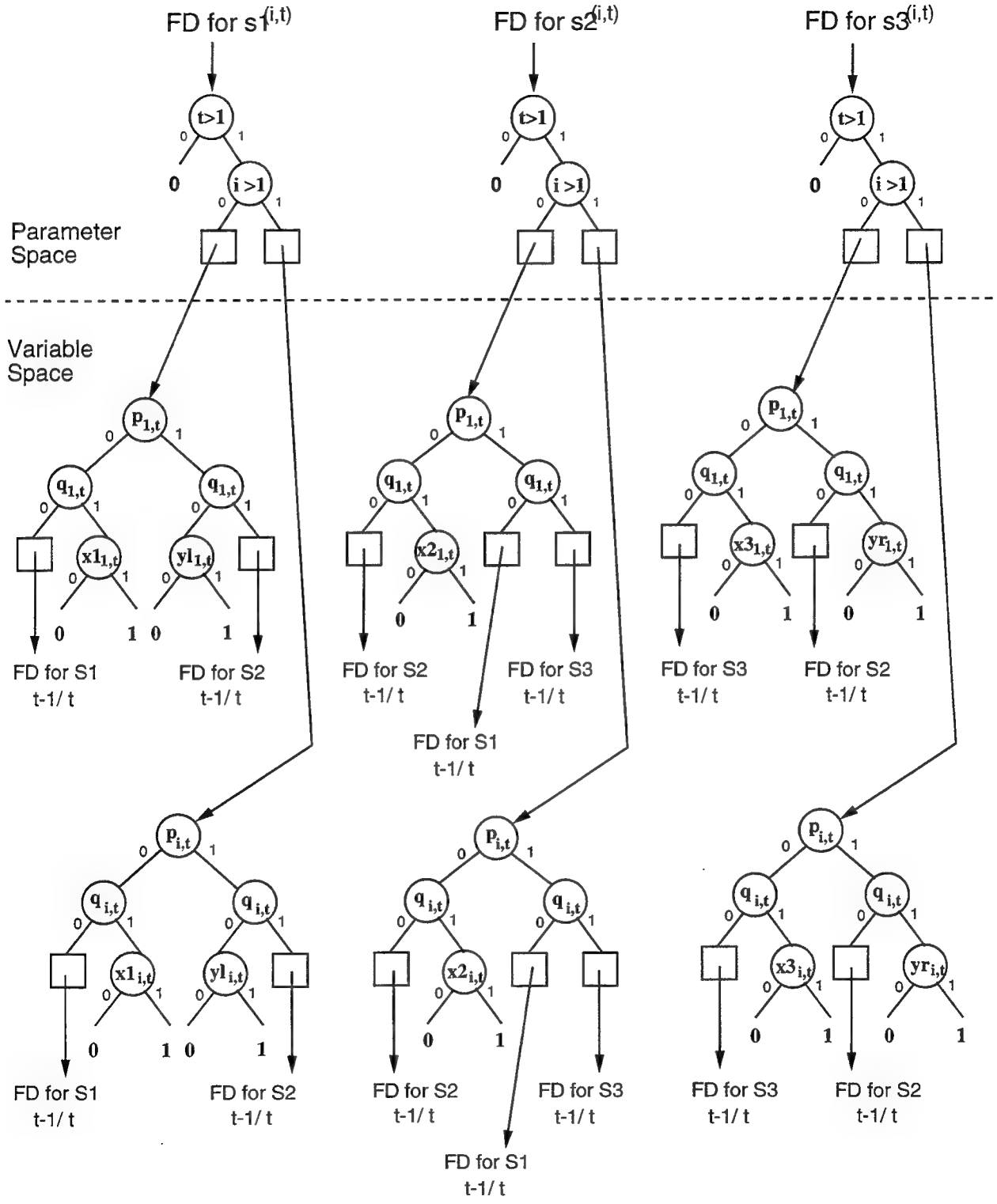


Figure 4.11: LIF Representation for Shift Register with Parametric Width

can be reused along any path through the IBF representation. This requirement is imposed in order to avoid the necessity of maintaining consistency between multiple uses of the same variable. Though existing techniques allow such multiple use of variables within the BDD context [87, 62], they result in significant overheads in symbolic manipulation. At this time, such use is not allowed within the IBF framework, though in principle it should not be difficult to incorporate it. Finally, note that the default variable ordering places the control variables (p, q) before the data variables $(x1, x2, x3, yl, yr)$.

4.3.2 Introduction of Dummy Parameters

Datapath circuits for data movement can be described inductively in terms of a length parameter also, which denotes the number of cells in the direction of data movement. However, typically there is an inter-dependence between the behavior of cells along the length. This leads to an inter-dependence between the time and space parameters, which prevents a direct IBF representation in these cases. However, if the inter-dependence has a very regular (inductive) pattern, an auxiliary technique involving introduction of dummy parameters can be used sometimes. This technique is best demonstrated with the following example.

Example 13: Consider another parametric version of the basic shift register (Example 10), which is now parametric in length, as shown in Figure 4.12. (In order to keep the exposition simple, the width parameter i is not considered in this example, though the following description holds for that too.) The natural inductive description for an n-length circuit can be easily given in terms of the space and time parameters (j and t, respectively) as:

for $t = 1, 1 \leq j \leq n$, $s^{(j,1)} = 0$

for $t > 1, j = 1$, $\quad s^{(1,t)} = (\neg p_t \wedge \neg q_t) \wedge s^{(1,t-1)} \vee (\neg p_t \wedge q_t) \wedge x_{(1,t)} \vee (p_t \wedge \neg q_t) \wedge s^{(2,t-1)} \vee (p_t \wedge q_t) \wedge yl_t$

for $t > 1, j = n$, $\quad s^{(n,t)} = (\neg p_t \wedge \neg q_t) \wedge s^{(n,t-1)} \vee (\neg p_t \wedge q_t) \wedge x_{(n,t)} \vee (p_t \wedge \neg q_t) \wedge yr_t \vee (p_t \wedge q_t) \wedge s^{(n-1,t-1)}$

for $t > 1, 1 < j < n$, $s^{(j,t)} = (\neg p_t \wedge \neg q_t) \wedge s^{(j,t-1)} \vee (\neg p_t \wedge q_t) \wedge x_{(j,t)} \vee (p_t \wedge \neg q_t) \wedge s^{(j+1,t-1)} \vee (p_t \wedge q_t) \wedge s^{(j-1,t-1)}$

It is easy to see from the last definition that the inter-dependence between cells is such that the j^{th} cell requires interaction with both the $(j-1)^{th}$ and the $(j+1)^{th}$ cells. This corresponds to a *bidirectional flow* of signals, unlike all previous examples which required a *unidirectional flow*. Note also that the inductive definitions for $t > 1$ are different for the end cells ($j = 1$ and $j = n$), where each uses its own data input (yl and yr, respectively).

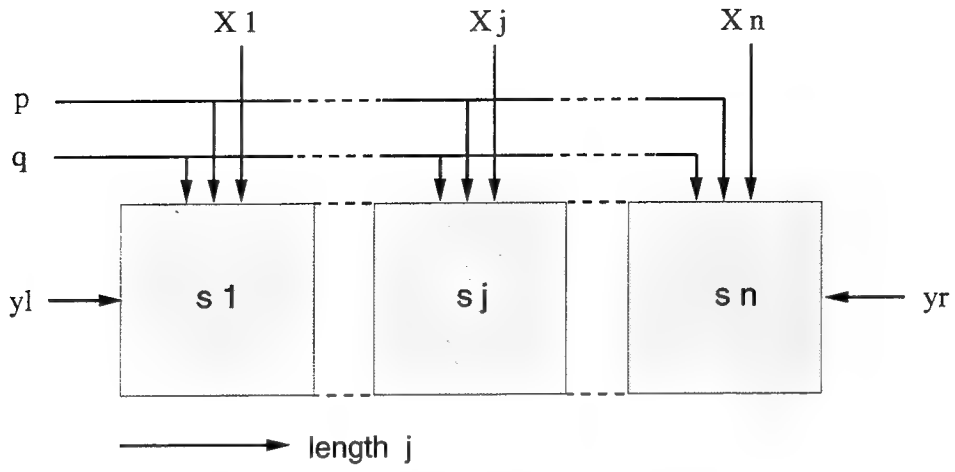


Figure 4.12: Shift Register with Parametric Length

As mentioned earlier, LIFs can naturally capture unidirectional iterative arrays, where the i^{th} cell depends inductively upon the $(i-1)^{th}$ cell. However, bidirectional flow cannot be captured inductively in terms of a single parameter. Therefore, a natural alternative is to introduce a dummy parameter, whereby bidirectional flow can be regarded as two flows in opposing directions, each of which can be captured inductively. The requirement, of course, is to ensure that the parameters interact according to the restrictions placed on the class of multiple parameter LIFs.

For the shift register example, introduce a dummy parameter k, which denotes the flow of signals in a direction opposite to that of j, as shown in Figure 4.13. Each cell is now parameterized in space by the pair (j,k), where the two end cells independently correspond to the basis case for each parameter. The behavioral description of this n-length circuit can now be rewritten as follows:

$$\text{for } t = 1, j \geq 1, k \geq 1,\ s^{(j,k,1)} = 0$$

$$\text{for } t > 1, j = 1, k = n,\ s^{(1,n,t)} = (\neg p_t \wedge \neg q_t) \wedge s^{(1,n,t-1)} \vee (\neg p_t \wedge q_t) \wedge x_{(1,t)} \vee (p_t \wedge \neg q_t) \wedge s^{(2,n-1,t-1)} \vee (p_t \wedge q_t) \wedge yl_t$$

$$\text{for } t > 1, j = n, k = 1,\ s^{(n,1,t)} = (\neg p_t \wedge \neg q_t) \wedge s^{(n,1.t-1)} \vee (\neg p_t \wedge q_t) \wedge x_{(n,t)} \vee (p_t \wedge \neg q_t) \wedge yr_t \vee (p_t \wedge q_t) \wedge s^{(n-1,2,t-1)}$$

$$\text{for } t > 1, 1 < j, k < n,\ s^{(j,k,t)} = (\neg p_t \wedge \neg q_t) \wedge s^{(j,k,t-1)} \vee (\neg p_t \wedge q_t) \wedge x_{(j,t)} \vee (p_t \wedge \neg q_t) \wedge s^{(j+1,k-1,t-1)} \vee (p_t \wedge q_t) \wedge s^{(j-1,k+1,t-1)}$$

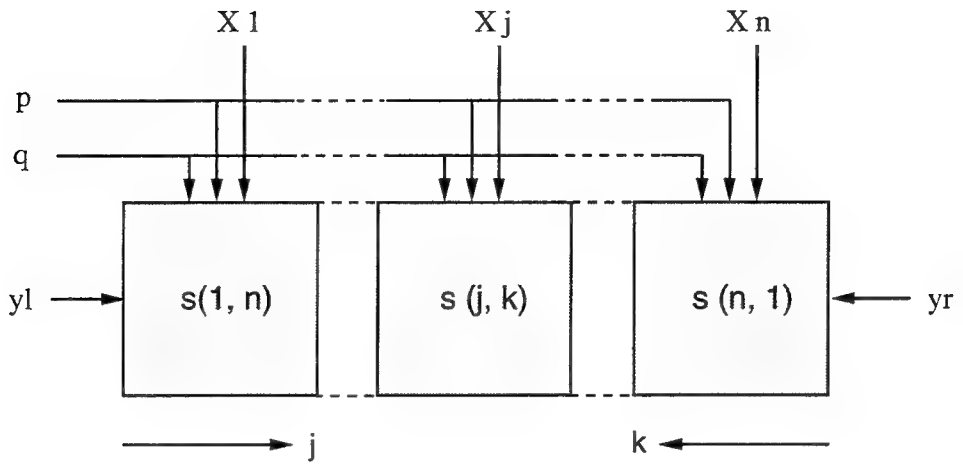


Figure 4.13: Shift Register with Dummy Length Parameter

The LIF representation for this description is shown in Figure 4.14. There are several points of interest in this representation. First, note that each pointer from an LIBDD terminal node to an FD does not represent a simple parameter substitution as before. In fact, it can be directly observed from the definitions that a j-instance LIF even refers to a $(j+1)$-instance LIF. However, note that in each such case, both other parameters (k and t) correspondingly decrease. In other words, the measure $j+k+t$ on parameters decreases on each inductive LIF reference. This property is crucial in generalizing the inductive argument on a single parameter which was used to establish the correctness of the LIF equality checking algorithm (Theorem 2, Chapter 3). Recall that by use of the EQ algorithm (Figure 3.9, Chapter 3), a Comparison-graph is constructed which contain an edge corresponding to each pointer from an LIBDD terminal node to an FD. The implicit parameter substitution $(i-1)/i$ on each such edge provides the inductive argument for resolving cycles in the graph. As a matter of fact, any well-founded ordering[3] on parameters can be used in its place. For the shift register example here, this ordering is provided by the measure $j+k+t$, which decreases along each pointer from an LIBDD terminal node to an FD. Interestingly enough, note that it is the ability to include time (t) as a separate parameter that provides this well-founded ordering. A simple next-cycle (combinational) behavior of the shift register would only provide $j+k$, which does not decrease with each inductive reference.

As a second interesting point, note that for an arbitrary n-length shift register circuit, only those representations of $s^{(j,k,t)}$ are meaningful (physically realizable) where $j+k+1=n$. This is because the dummy parameter k is not an independent parameter of description, but has a fixed relationship with the original parameter j. However, it is possible to view the parameters as

[3]A *linear* ordering is defined by an order relation "<" which is irreflexive, transitive and total. A *well-founded* ordering is a linear ordering such that each nonempty set possesses a least element.

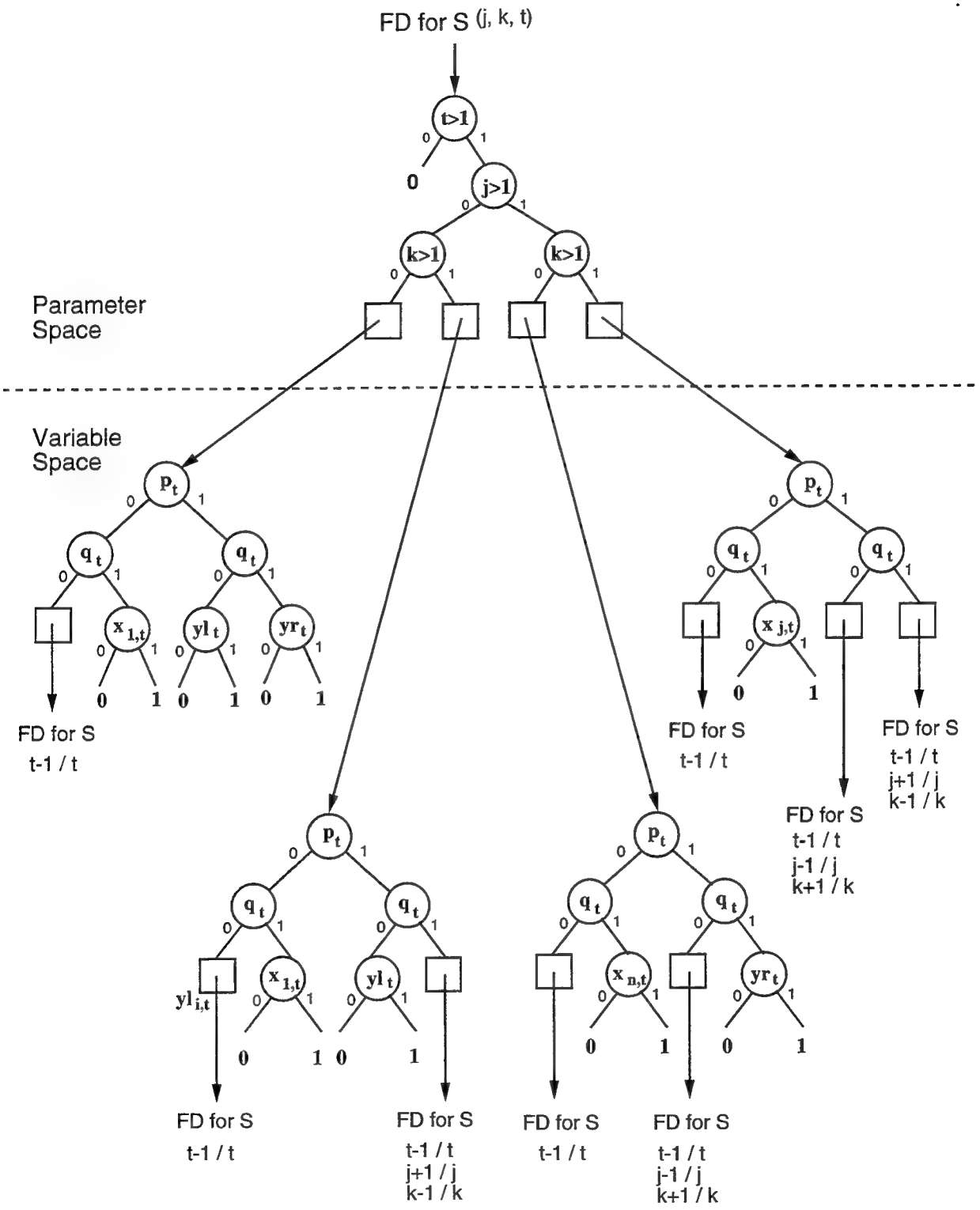


Figure 4.14: LIF Representation for Shift Register with Parametric Length

being independent for the purpose of the LIF representation. In this view, the restriction should be accounted for separately, if desired, by any applications that may utilize this representation. This is reminiscent of a similar technique used by Burch for OBDD representation of a multiplier in verification applications [33]. Though a direct OBDD representation of a multiplier is known to be exponential in the size of the circuit inputs [26], Burch obtained a polynomial-sized representation by introduction of extra (dummy) variables that denote multiple fan-ins of the inputs. The restriction, that each set of variables corresponding to the fan-ins of the same input should be identical, is not enforced in the representation. Rather, it is left to the verification applications to account for this restriction. This has the advantage of greatly simplifying the verification, at least in those cases where the specification can be naturally expressed in terms of the split variables also. Similarly, though the LIF representation corresponding to the shift register description does not explicitly account for the restriction $j + k + 1 = n$, it is canonical within the given parametric framework. Furthermore, since the parameters j and k provide a very natural framework, it is quite likely that verification applications may use the same parameters to express the specifications also, thereby greatly simplifying the task of parametric verification.

In general, the technique of using a dummy parameter to denote an opposing direction of signal flow can be used in other settings as well. Potentially, any translation-independent bidirectional behavior can be parametrically represented. Within the IBF framework, the additional criterion is that the resulting inductive description should follow the limitations of the multiple parameter IBF representations. However, as other mechanisms are used to overcome these limitations, the application scope of this technique may also increase. One such example is described later in this chapter (Section 4.5), in conjunction with representation of multiple output circuits.

4.3.3 Introduction of Dummy Functions

It has been mentioned earlier that the primary reason for not handling simultaneous induction in the current multiple parameter framework is that it may require handling of Boolean combinations of functions with non-uniform parameter values. The difficulty in handling such Boolean combinations is illustrated by the following simple example of an accumulator circuit.

Example 14: Consider an i-bit accumulator circuit with parametric data input x, as shown in Figure 4.15. It is similar to the ripple carry adder, except that it has a memory element (shown as delay 'D' in the figure), which stores the current output and uses it as an input for the next time instant. The output of the i^{th} cell at time t, denoted $s^{i,t}$, can be defined inductively as follows:

for $t = 1, i \geq 1$, $s^{(i,1)} = x_{(i,1)}$
for $t > 1, i = 1$, $s^{(1,t)} = x_{(1,t)} \oplus s^{(1,t-1)}$

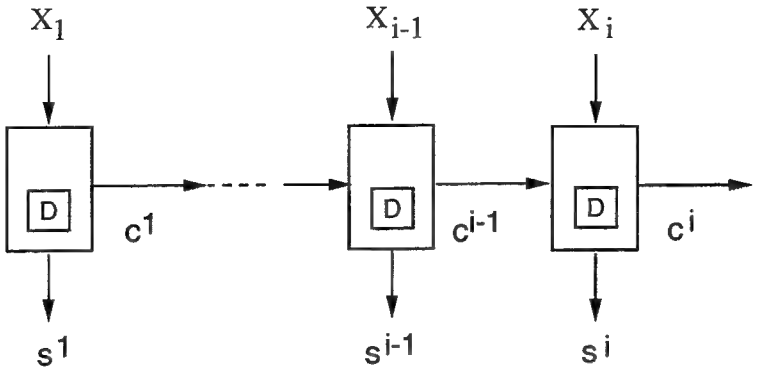


Figure 4.15: Parametric Description of an Accumulator

for $t > 1, i > 1$, $s^{(i,t)} = x_{(i,t)} \oplus s^{(i,t-1)} \oplus c^{(i-1,t)}$

where the function $c^{(i,t)}$ represents the carry function between accumulator cells, defined as:

for $t = 1, i = 1$, $c^{(1,1)} = 0$
for $t = 1, i > 1$, $c^{(i,1)} = x_{(i,1)} \wedge c^{(i-1,1)}$
for $t > 1, i = 1$, $c^{(1,t)} = x_{(1,t)} \wedge s^{(1,t-1)}$
for $t > 1, i > 1$, $c^{(i,t)} = (x_{(i,t)} \wedge s^{(i,t-1)}) \vee ((x_{(i,t)} \vee s^{(i,t-1)}) \wedge c^{(i-1,t)})$

Recall that when inductive definitions involve Boolean combinations of functions with uniform parameter values, a new (dummy) function is introduced to denote the Boolean combination. Here, the definitions of functions s and c of the accumulator example involve Boolean combinations of functions with non-uniform parameter values, e.g. $s^{(i,t-1)} \oplus c^{(i-1,t)}$, and $s^{(i,t-1)}) \wedge c^{(i-1,t)}$, respectively. Suppose a similar technique is used, where a dummy function, F, is introduced to denote such a combination:

$$
\begin{aligned}
F^{(i,t)} &= s^{(i,t-1)}) \wedge c^{(i-1,t)} && \text{by definition} \\
&= (x_{(i,t-1)} \oplus s^{(i,t-2)} \oplus c^{(i-1,t-1)}) \wedge && \text{by substitution} \\
&\quad ((x_{(i-1,t)} \wedge s^{(i-1,t-1)}) \vee ((x_{(i-1,t)} \vee s^{(i-1,t-1)}) \wedge c^{(i-2,t)}) && i > 2, t > 2
\end{aligned}
$$

Note that the latter definition now requires a Boolean combination of $s^{(i,t-2)}$ and $c^{(i-2,t)}$. If this Boolean combination is denoted by another dummy function, then its parametric definition would further require a Boolean combination of $s^{(i,t-3)}$ and $c^{(i-3,t)}$, and so on. This process would go on indefinitely, and an infinite series of new functions will be required. Thus, it is not possible to capture the sequential behavior of an i-bit accumulator in the form of a finite set of IBFs.

On the other hand, the technique of introducing dummy functions motivates a slightly different goal. By introducing a new function (LIF) to denote the sequential behavior of an i-bit

accumulator after each time instant, it is possible to use function (LIF) composition techniques to capture a fixed-length sequential behavior of the accumulator. This is similar in concept to fixed-length *symbolic simulation* of circuits [24], applied here in the context of parametric circuits. Note that this also involves introduction of new symbolic variables representing the inputs at each time instant. For example, the LIF representation for the sequential behavior of the i-bit accumulator after $t = 3$ is shown in Figure 4.16, where:

- Data inputs $x1_i, x2_i, x3_i \ldots$ denote data inputs to an i-bit accumulator at times instants $t = 1, 2, 3 \ldots$, respectively.
- LIFs $s1^i, s2^i, s3^i \ldots$ denote the sum outputs of the i^{th} cell of the accumulator at time instants $t = 1, 2, 3 \ldots$, respectively.
- LIFs $c1^i, c2^i, c3^i \ldots$ denote the carry outputs from the i^{th} cell of the accumulator at time instants $t = 1, 2, 3 \ldots$, respectively.

Note that only parameterization in space (i) is used for both the LIFs and the symbolic variables, while the time dimension has been made explicit. In Chapter 6, yet another aspect of this technique will be highlighted, with respect to its applications in formal verification.

4.4 Representation of Multiple Circuit Outputs

For all parametric circuit examples considered so far, the outputs of an i-instance circuit have been represented either as i-instance IBFs (e.g. comparator, serial parity), or as the set of all j-instance IBFs, $1 \leq j \leq i$ (e.g. ripple carry adder). In both cases, only a fixed number of i-instance IBFs are needed to directly capture all outputs of an i-instance circuit. Since the size of IBF representations is independent of the induction parameter, it ensures that the circuit output representation is also independent of the parameter of description.

However, for many parametric circuits, the number of outputs of an i-instance circuit is a non-trivial function of the induction parameter i. In such cases, it is not desirable to represent each output explicitly as an IBF, since it would lead to a circuit representation where the size depends on the induction parameter. Fortunately, there is usually some kind of inductive structure among related multiple outputs, which can be exploited to parametrically represent the outputs by using an *output index*. Two different alternatives can be used for representation of an output index within the context of IBF schemata – it can represented as a parameter, or it can be encoded as an input argument for the function. These are described in detail in the following sections. Note, again, that both these alternatives can be applied in conjunction with any general schema for canonical representation of a single output parametric function.

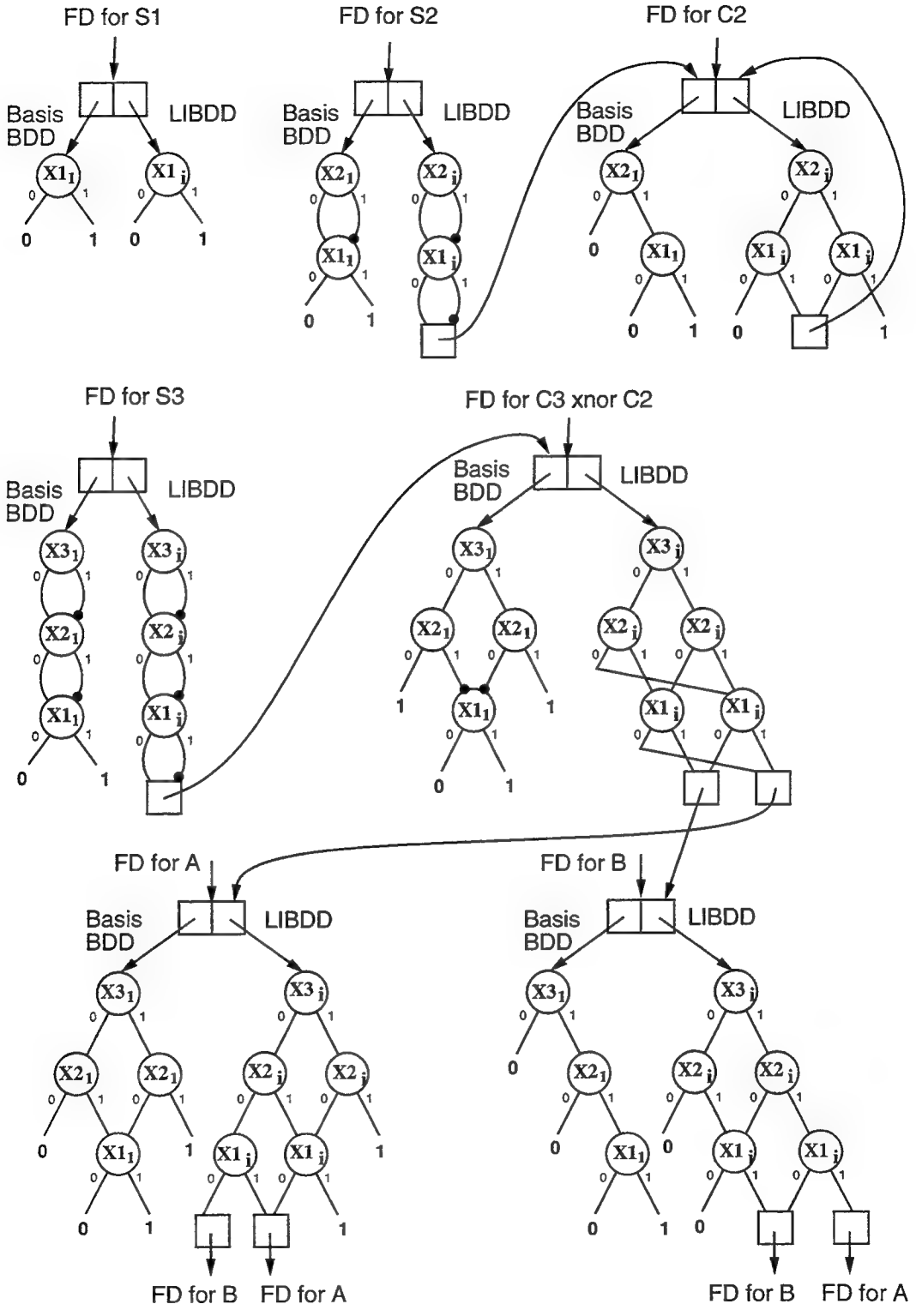


Figure 4.16: LIF Representations for Symbolic Simulation of an Accumulator

4.4.1 Output Index as a Parameter of Description

The output index can be represented as a separate parameter of the inductive circuit description. Typically, it is a *dependent* parameter, in the sense that its range of values, and different partitions in that range, are expressed as functions of other parameters such as circuit size. For example, consider a register file with i-bit address and j-bit data. A separate parameter k can be used as an output index, in order to access a particular read-port output from among the j read-port outputs. Here, i and j are regarded as inherent parameters of the parametric circuit description, while k is regarded as a dependent parameter with the range $1 \leq k \leq j$.

In this framework, a single circuit output is captured as the appropriate projection of a multiple parameter IBF in the geometric hyperspace defined by the output index parameter and other circuit parameters. The representation for this framework has already been described in Section 4.2, in the general context of multiple parameter IBFs. However, some extensions are needed in order to handle the range and partitions of an output index parameter. For such parameters, the non-terminal nodes of a parameter decision tree are now allowed to involve a comparison against hyperspace partitions. In fact, if comparison for inherent parameters is similarly extended, then arbitrary orthogonal trajectories (such as the dotted-line trajectory of Figure 4.7) can also be represented. However, for now, comparison of inherent parameters is allowed only against their basis values (as before). Note that since a basis value denotes a hyperplane, this can also be viewed as a comparison against a hyperspace partition. Thus, each path from the root of a parameter decision tree to a terminal node corresponds to a particular region of the geometric hyperspace, where each non-terminal node identifies a boundary of this region. Furthermore, the terminal node of the parameter decision tree for each path can be viewed as providing a representation of the function for that region.

For canonicity of this representation, note that within the IBF framework, the terminal nodes of the parameter decision tree point to Basis BDDs and LIBDDs/EIBDDs. These can be made canonical by using the IBF schemata described in Chapter 3. The remaining task is to represent the non-terminal nodes of the parameter decision tree in a canonical form. Again, this is done by imposing a total ordering on them, followed by BDD-style reductions to remove redundant nodes and isomorphic subgraphs. With the introduction of partitions, the task of redundant node removal now includes a check to ensure that each non-terminal node partitions the hyperspace into non-null regions. In other words, if a particular comparison results in a null region, then the associated non-terminal node is redundant. The final results is a maximally reduced DAG, which is canonical with respect to the ordering.

The basic idea for ordering non-terminal nodes is to utilize the user-given ordering on parameters (π, described in Section 4.2)[4], in conjunction with some scheme to order the partitions for each dependent parameter. By restricting comparisons for dependent parameters to those against the set of totally ordered partitions, a total ordering on all non-terminal nodes is obtained. The

[4]The convention is to place all inherent parameters above the dependent parameters.

essential requirement, therefore, is a scheme for ordering partitions. As mentioned earlier, both the range and partitions for output index parameters are usually specified as functions of size parameters of the circuit. For simplicity of analysis, only *linear partitions* in an n-parameter hyperspace are considered, i.e. each hyperplane partition is characterized by a linear equation (with real coefficients) in n parameters. Two different schemes have been explored for ordering such linear partitions, described in the following subsections. They are based on whether or not intersection of hyperplanes is allowed, since this distinction affects the task complexity to a great extent.

4.4.1.1 Non-intersecting Hyperplanes Ordering Scheme

In this scheme, the restriction is that partition hyperplanes for all parameters should not intersect in the *allowable* hyperspace, where the allowable hyperspace is defined as the region bounded by the range of each parameter. This allows a simple criterion for ordering the partitions for each dependent parameter, based on the range of integral values it can take on either side of the partition. (The criterion is easy to formulate by using other equivalent geometric information such as slopes and intercepts.) For example, in a 2-dimensional parameter space defined by $i \geq 1$, $1 \leq j \leq 2 \times i$), consider two partitions $A : (j = i/2)$ and $B : (j = i)$ as shown in Figure 4.17 Part(a). These hyperplanes are non-intersecting in the allowable hyperspace, shown as the shaded region. Therefore, they can be totally ordered according to the values j can take. Since the range allowed by A is less than that allowed by B, $A < B$ in the partition ordering, leading to a parameter decision tree as shown in Part (b) of the same figure.

Once the partitions have been ordered, a lexicographic ordering is used on the pair *(parameter, partition)*, to order the non-terminal nodes. In other words, given parameters i, j and partitions p, q, a non-terminal node with comparison "$(i > p)$" is placed before the non-terminal node with comparison "$(j > q)$", if and only if

$$(i <_\pi j) \vee ((i = j) \wedge (p <_\alpha q))$$

where "$<_\pi$" represents the user-given ordering on parameters, and "$<_\alpha$" represents the partition ordering obtained by geometric analysis of the partition equations.

A useful property of two non-intersecting hyperplanes is that one of the two regions resulting from one hyperplane partition will always be entirely contained within a region resulting from the other. This is very important in practice, because it allows detection of null regions using the same criterion that is used for partition ordering. As mentioned earlier, detection of null regions is essential for removal of redundant non-terminal nodes in the parameter decision tree. Furthermore, this property also ensures that there are a linear number of non-null regions, in terms of the number of partitions, resulting in a linear-sized representation.

As a practical example of the multiple output representation, consider a barrel shifter circuit [111] with the following parametric description:

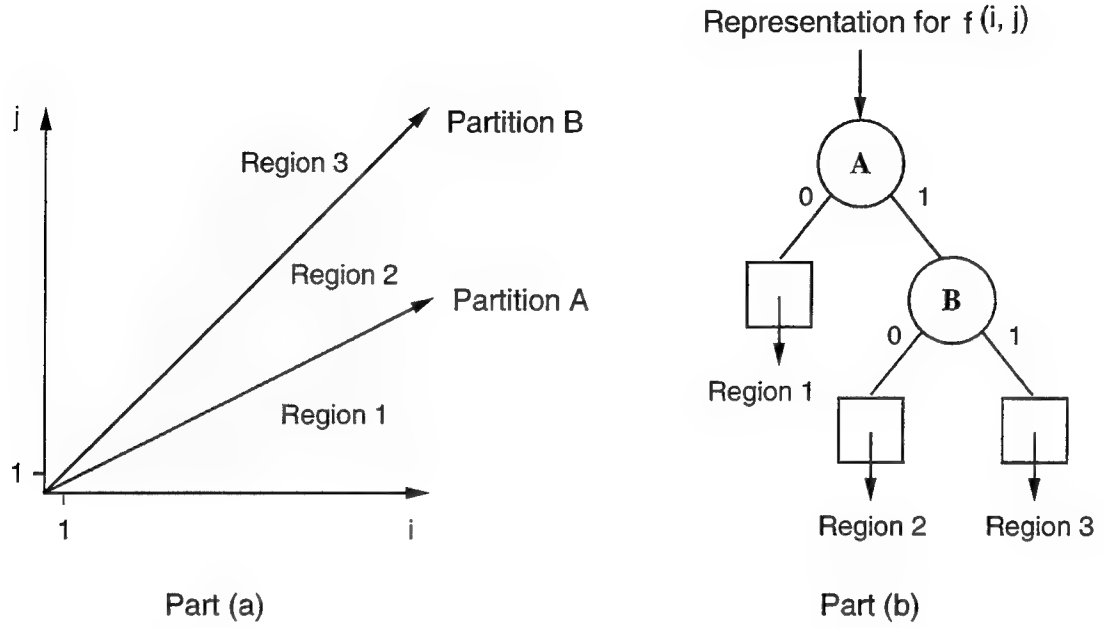


Figure 4.17: Non-Intersecting Partitions in Parameter Hyperspace

Example 15: An i-instance unidirectional barrel shifter has two parametric inputs — $X[2i-1]$ denoting the data to be shifted, and $S[i]$ denoting the desired shift (where $S_i = 1$ for a shift by $i-1$). As an illustration, the instance for $i = 4$ is shown schematically in Figure 4.18, where the transmission gate implements the transmission of x to bsh if s is on (If s is off, bsh is floating). Note from the figure, that an i-instance circuit has i outputs, where all but the i^{th} one are derived by some *new computation* on the outputs of the $(i-1)$-instance circuit. This implies that unlike the example of the ripple carry adder, outputs from the $(i-1)$-instance circuits cannot be directly used as outputs from the i-instance circuit. Thus, it is not possible to use the circuit size parameter (i) to index the multiple outputs. Instead, a separate parameter j is used as an output index, where $1 \leq j \leq i$. The parametric description for the outputs $bsh^{(i,j)}$ can now be given as follows:

for $i = 1, j = 1$, $\quad bsh^{(1,1)} = S_1 \wedge X_1$
for $i > 1, 1 \leq j < i$, $bsh^{(i,j)} = (S_i \wedge X_{i+j-1}) \vee bsh^{(i-1,j)}$
for $i > 1, j = i$, $\quad bsh^{(i,i)} = (S_i \wedge X_i) \vee select^{(i-1)}$

where $select$ is a new function, defined as

for $i = 1$, $\quad select^1 = S_i \wedge X_i$

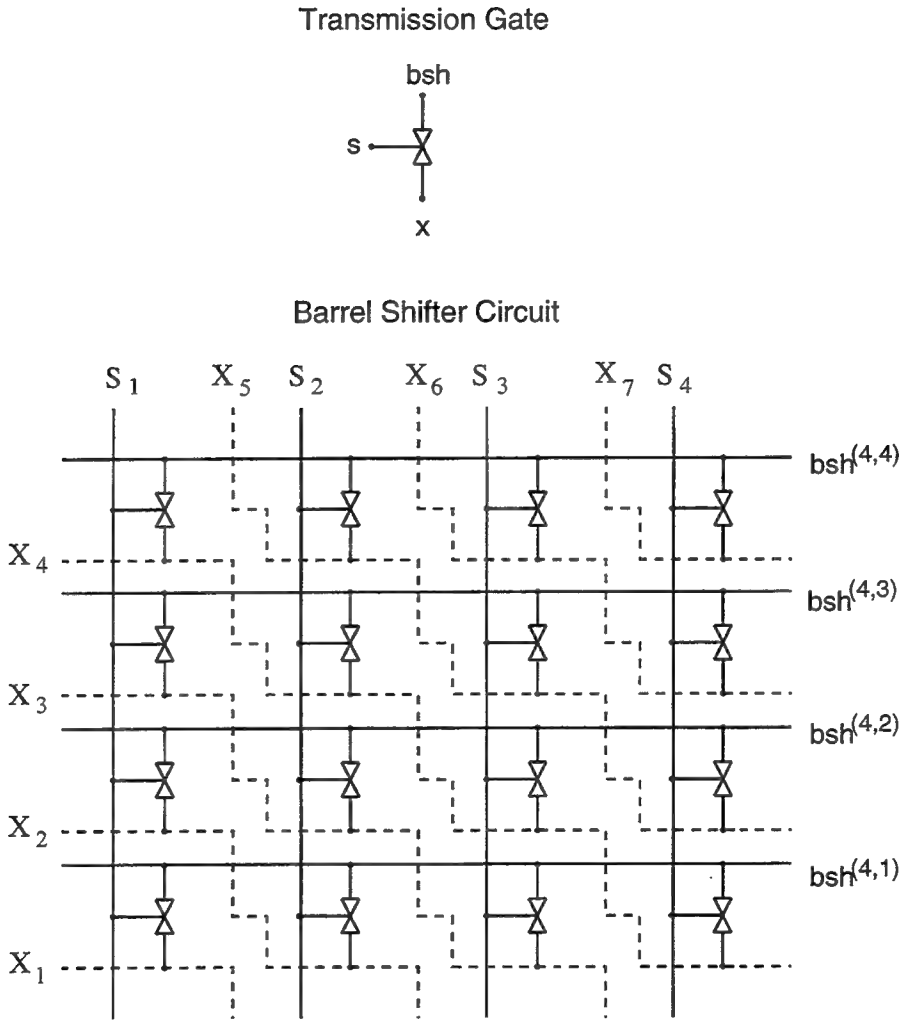


Figure 4.18: A Barrel Shifter Circuit

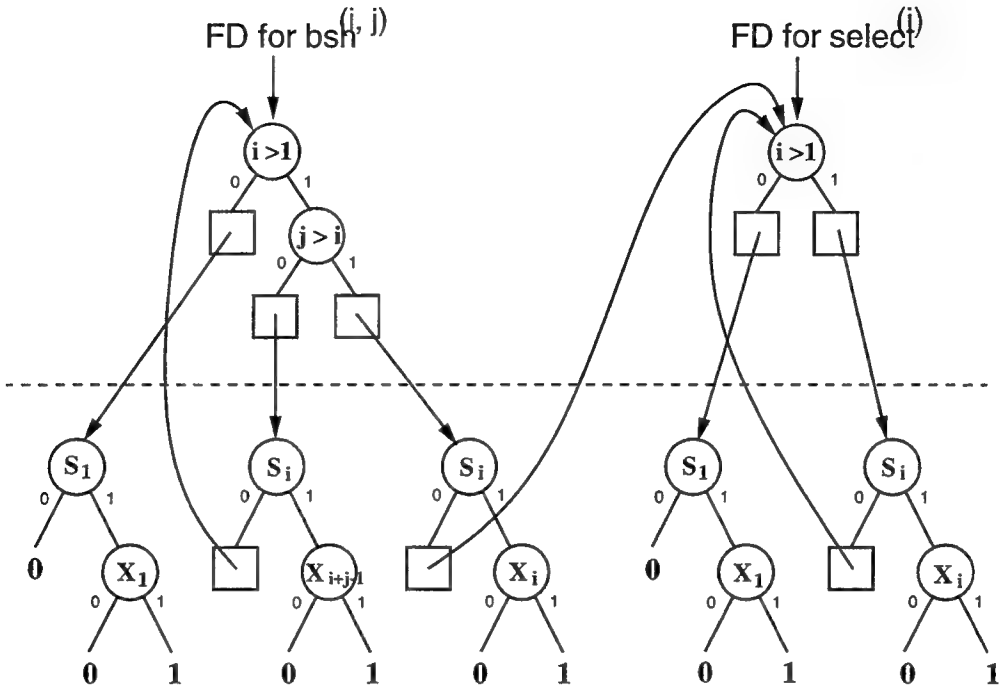


Figure 4.19: LIF Representation for a Barrel Shifter Circuit

for $i > 1$, $\quad select^i = (S_i \wedge X_i) \vee select^{i-1}$

Since the partitions for the parameters i ($i = 1$) and j ($j = 1, j = i$) are non-intersecting in the allowable hyperspace, the scheme outlined in this section can be used to represent the multiple outputs of the i-instance circuit. This representation is shown in Figure 4.19, where parameter ordering $i <_\pi j$ is used, and the partitions for j are ordered according to the range of j values.

4.4.1.2 Intersecting Hyperplanes Ordering Scheme

When partition hyperplanes intersect in the allowable hyperspace, a different scheme needs to be used. Though no practical examples of inductive hardware circuits which necessitate this scheme have emerged so far, the following description is provided for the sake of completion. As before, note that it can be used for handling multiple parameter functions in general.

Consider a typical example with two parameters i and j, and two intersecting hyperplanes C and D, as shown in Figure 4.20 Part(a). Unlike the case of non-intersecting hyperplanes, note that there is no clear choice in this case regarding the partition ordering. Furthermore, no matter which ordering is chosen, the complete decision tree on the two partitions is needed in order to

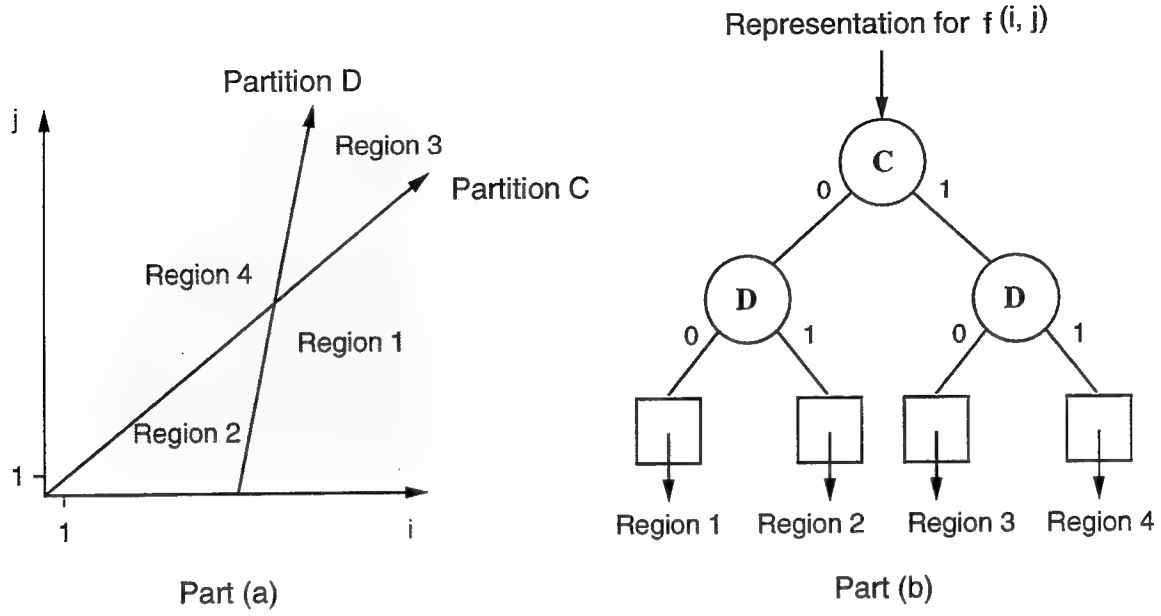


Figure 4.20: Intersecting Partitions in Parameter Hyperspace

represent all four regions of the hyperspace, as shown in Part (b) of the same figure. In general, the number of regions is exponential in the number of partitions.

The requirements, again, are to provide a fixed ordering for the partitions, and to identify paths in the parameter decision tree that correspond to non-null regions. Any ordering based strictly on the syntactic representation of the hyperplane equations can be used to order the partitions. However, the same criterion cannot be used to detect null regions as before. In the general case, the problem of identifying non-null regions bounded by hyperplanes is of non-trivial complexity, and a standard linear programming (LP) package can be used to handle this task. (Note that when an LP problem solution indicates a non-null region, it is possible that no *integer value* is contained in that region. However, this situation is unlikely to occur in typical cases, and will affect only the efficiency of the representation, not the correctness.) Most available LP implementations are of exponential complexity, but are fast in practice. (Polynomial time algorithms, though known, are difficult to implement.) Another alternative is to consider the special cases for 2-dimensional and 3-dimensional space, and explicitly solve for a region bounded by n hyperplanes. The complexity of these problems is $O(n \log n)$ [126]. These special cases will most likely be adequate for handling circuits in practice.

4.4.2 Output Index as a Binary-encoded Argument

When an output index does not bear a simple linear relationship with other parameters of the circuit description, it is not possible to use the schemes described so far. A different technique is to encode the output index as a parameterized Boolean-valued argument for the function. In this respect, it is treated identically as a parameterized Boolean-valued circuit input, along with the same syntactic restrictions for the IBF classes. A typical practical application is in handling an i-instance circuit with 2^i outputs, where an i-bit argument encodes the output index for an i-instance IBF.

Example 16: Consider an i-instance decoder with 2^i outputs, and a parametric input $A[i]$. Each of the 2^i outputs can be indexed by using an i-bit Boolean-valued argument $J[i]$, where the binary value represented by the i bits is used to index the range from 0 to $(2^i - 1)$. Furthermore, the decoder circuit can be described parametrically in such a way that the outputs for the i-instance circuit can be obtained in terms of the i^{th} bit of J, and the outputs from $(i-1)$-instance circuits. This is shown schematically in Figure 4.21, and is described as follows:

$$\begin{aligned}
\text{for } i = 1 : dec^1(A_1, J_1) &= (\neg J_1 \wedge \neg A_1) \vee (J_1 \wedge A_1) \\
\text{for } i > 1 : dec^i(A[i], J[i]) &= (\neg J_i \wedge \neg A_i \wedge dec^{i-1}(A[i-1], J[i-1])) \vee \\
&\quad (J_i \wedge A_i \wedge dec^{i-1}(A[i-1], J[i-1])).
\end{aligned}$$

Note that the i^{th} bit of J (denoted J_i) is used for choosing the appropriate half in the range of J ($0 \leq J < 2^i$), and the remaining $(i-1)$ bits of J (denoted $J[i-1]$) are passed on as an implicit argument to the $(i-1)$-instance function dec^{i-1}. This exactly matches the requirements for an LIF description. The corresponding LIF representation is shown in Figure 4.22.

This technique is particularly useful for circuits that are organized inductively in the form of a tree, e.g. prefix parity circuit, carry outputs of a carry lookahead adder etc. Essentially, the vector of outputs at depth i can be divided into halves, such that the j^{th} output within each half depends upon the j^{th} output at depth $(i-1)$. It is the binary-encoding of the output index that allows the exponential nature of a tree circuit to be captured in a linear form.

4.4.3 Handling Vectors of Outputs

The focus of the previous subsections has been on representation of an individual output of a multiple output circuit, where the range of the output index implicitly defines a vector of outputs. However, in some cases, a representation for an explicit vector (or subvector) of outputs may be required. Such a capability is particularly useful in handling compositions of inductively-defined hardware units, where an output vector from one unit is used as the input vector for another unit.

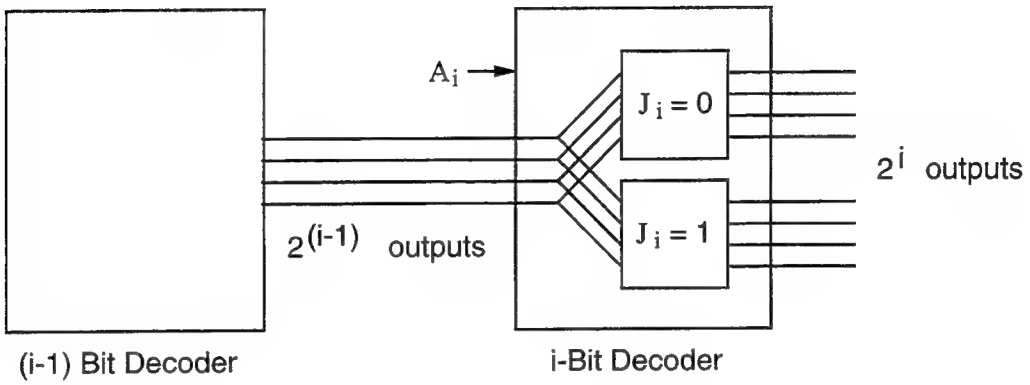


Figure 4.21: Parametric Description of a Decoder

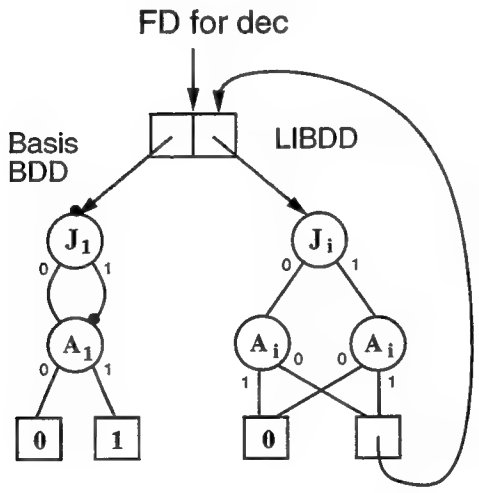


Figure 4.22: LIF Representation for Decoder Outputs

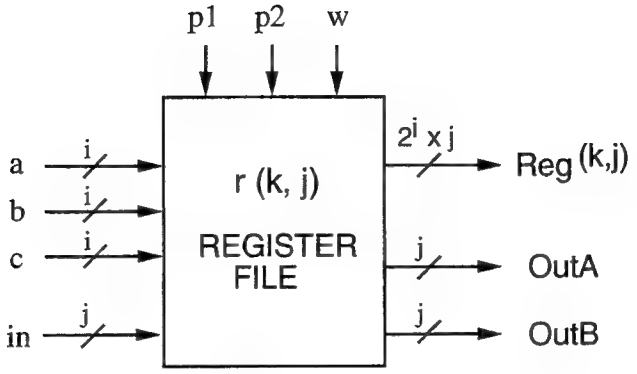


Figure 4.23: Parametric Description of a Register File

Recall that subvectors of inputs (for representing parametric inputs) are specified by using a length/offset notation (Section 4.1.1). The same approach can be used to denote subvectors of outputs also, i.e. a subvector of outputs is defined in terms of its length, and offset of its first element with respect to the underlying output vector. Both of these components – length, and offset – can be regarded as separate parameters of the inductive description. Furthermore, an explicit subvector index can be used to represent each output within the range of the subvector. These are represented using the same techniques that have been described for representation of multiple parameters and multiple outputs.

4.5 Combining Different Techniques: Examples

The different representation techniques described in this chapter are not mutually exclusive. Rather, they can be frequently combined in order to handle relatively more complex practical circuits. This is best illustrated with the following examples.

Example 17: Consider a space-parametric description of a 3-port register file, shown schematically in Figure 4.23. It has i-bit address lines – denoted by $a[i]$ and $b[i]$ for two read-ports, and $c[i]$ for a write-port. There are j-bit data lines – $OutA$, $OutB$ denoting outputs of the read-ports (addressed by a, b, respectively), and in denoting the data input on the write-port.

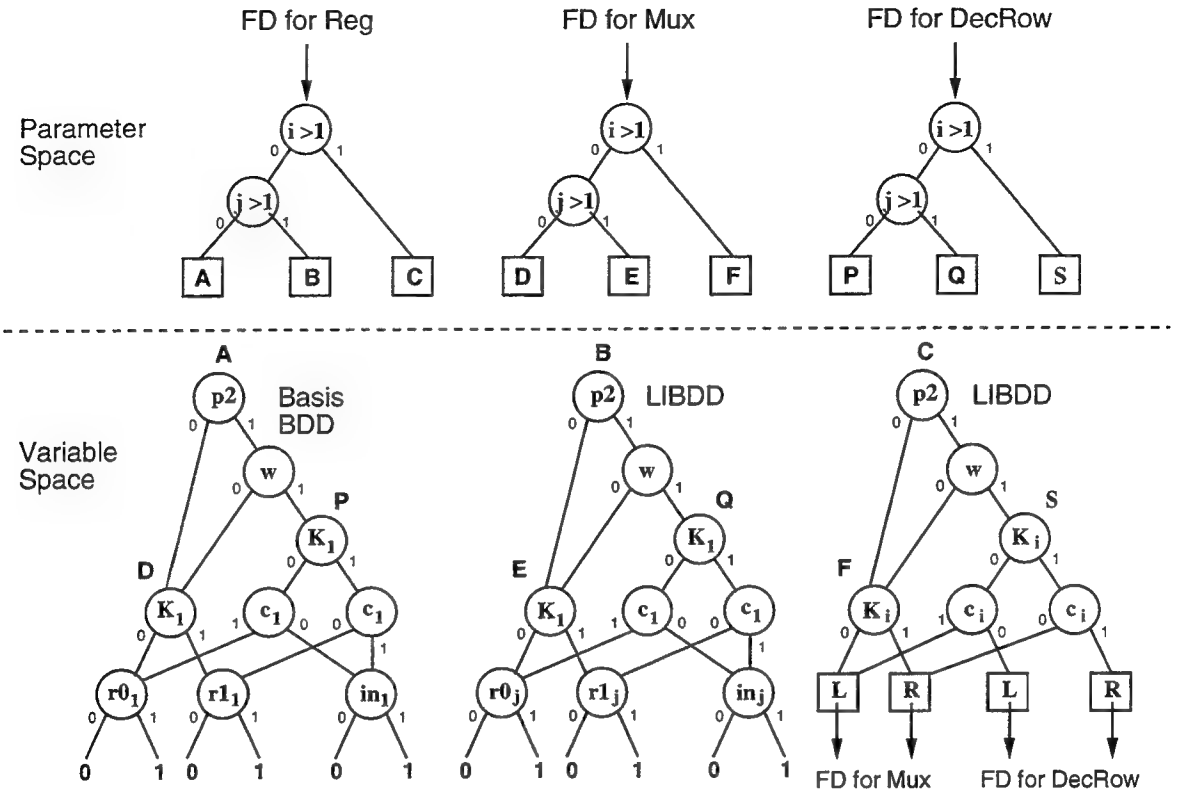


Figure 4.24: LIF Representation for Register File Cells

The control signals $p1, p2$ denote the clock signals for a standard 2-phase clock, and w denotes the write signal.

The new register file contents (after a clock cycle) can be described by a multiple output LIF $Reg^{(i,j)}$, where the 2^i rows are indexed by using a binary-encoded i-bit argument $K[i]$, and a particular bit in each row is indexed by the parameter j itself. The LIF representation for $Reg^{(i,j)}$ is shown in Figure 4.24, where $r[2^i, j]$ denotes the parametric inputs representing the old register cell contents. Note that while the parameters i and j belong to the parameter space, the Boolean-valued argument $K[i]$ used to index outputs belongs to the variable space.

Similarly, the outputs of the read-ports can also be represented as multiple parameter LIFs, where the parameter j is used directly to index the j-bits of the read-port. The LIF representation for the read-port output $OutA^{(i,j)}$ is shown in Figure 4.25. Note that the parametric address input $a[i]$ is used to select the appropriate row (amongst the 2^i rows of the register file) in a manner identical to the output index argument $K[i]$ in the representation for Reg.

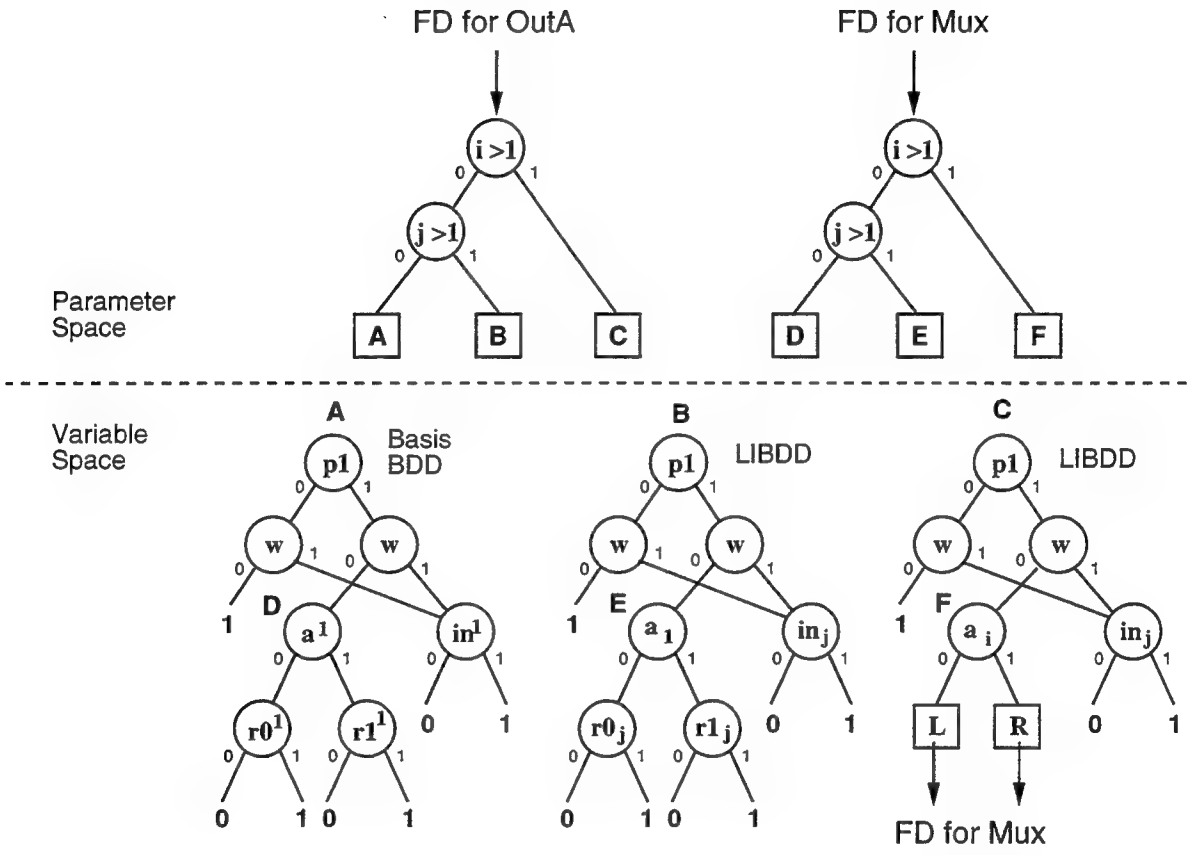


Figure 4.25: LIF Representation for Register File Outputs

The final example consists of a popular benchmark circuit called MINMAX [148]. Though the circuit was originally presented in the context of sequential behavior verification, it has been subsequently used by many researchers to benchmark other applications also. Here, a space-parametric combinational version of the circuit is considered, as follows:

Example 18: An i-instance MINMAX circuit is shown schematically in Figure 4.26. It consists of two i-bit registers ($L[i]$ and $H[i]$), two i-bit comparators ($Comp1$ and $Comp2$), two i-bit multiplexors ($Mux1$ and $Mux2$), and an i-bit ripple carry adder ($Adder$). The circuit has an i-bit data input $X[i]$, three control inputs c (clear), e (enable), and r (reset), and i-bit data outputs Out. The behavioral operation of the circuit is as shown in Figure 4.27 (adapted from another paper [56]).

In order to represent a one-cycle (combinational) behavior of this circuit, the goal is to represent the circuit outputs as multiple parameter IBFs. Though the entire circuit has an intuitive parametric flavor, and is indeed constructed from inductively-defined units, yet the circuit outputs do not have an obvious inductive description. The reason is that the comparator outputs ($Comp1, Comp2$) can affect the multiplexor selection inputs ($Mux1, Mux2$, respectively) in such a way that the j^{th}-bit of the output can depend upon the k^{th}-bit of the data input, where $1 \leq j < k \leq i$. Such a dependence is not allowed by either of the IBF classes.

However, upon closer examination, it can be seen that this troublesome dependence is also very structured. In fact, the comparator operation affects the j^{th}-bit of the output only from those k-bits of the input, where $1 \leq j \leq k \leq i$. In other words, the comparator operation involves a flow of information in a direction opposite to i. As described in Section 4.3.2, bidirectionality can be handled in many cases by introducing a dummy parameter. Therefore, a dummy parameter can be introduced to capture the comparator operation.

On the other hand, note that the adder operation affects the j^{th}-bit of the output from those k-bits of the input where $1 \leq k \leq j \leq i$. The interesting observation is that this set of inputs (apart from the j^{th}-bit itself) is mutually exclusive from the set involved with the comparator operation! In this sense, the dummy parameter j actually pinpoints the break-point of influence of the higher- and lower-instance inputs for every j-bit of the output. Furthermore, it can be directly used as an output index also, where $1 \leq j \leq i$.

An LIF representation for $Out^{(i,j)}$ can now be given as shown in Figure 4.28. In this figure, the auxiliary LIFs used by the terminal nodes of the LIBDDs are not shown. Essentially, they are used to determine the final two arguments to the adder, at whatever point it is possible to do so. There are two sets of LIFs – Sum functions and Carry functions. The former are denoted $S**$, and the latter $C**$, where each '$*$' is one of:

- X – denoting that the argument is known to be the data input X

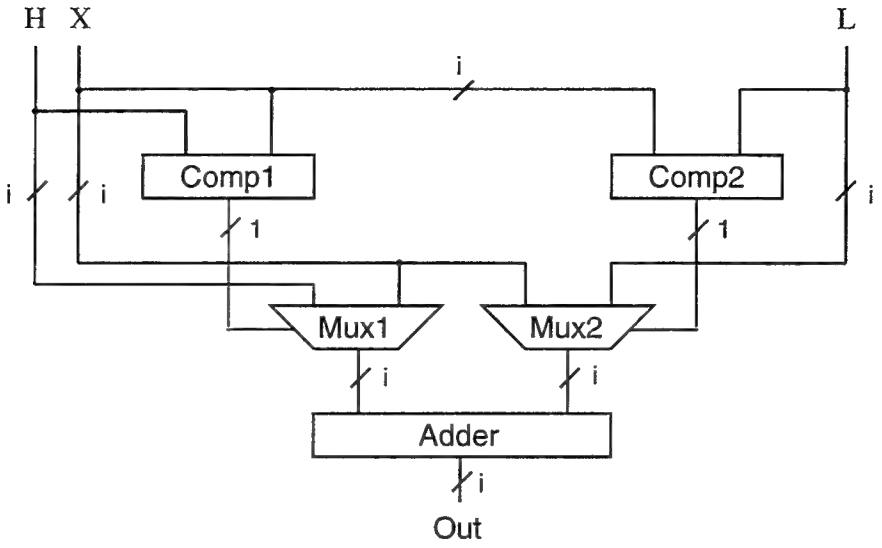


Figure 4.26: Parametric Description of the MINMAX Circuit

$$Out^t = (L^t + H^t) >> 1$$

$$L^t = \begin{array}{l} \text{if } c_t \text{ then } 0 \\ \text{else if } NOT\ e_t \text{ then } L^{(t-1)} \\ \text{else if } r_t \text{ then } X_t \\ \text{else } minimum(X_t, L^{(t-1)}) \end{array}$$

$$H^t = \begin{array}{l} \text{if } c_t \text{ then } 0 \\ \text{else if } NOT\ e_t \text{ then } H^{(t-1)} \\ \text{else if } r_t \text{ then } X_t \\ \text{else } maximum(X_t, H^{(t-1)}) \end{array}$$

Figure 4.27: Operation of the MINMAX Circuit

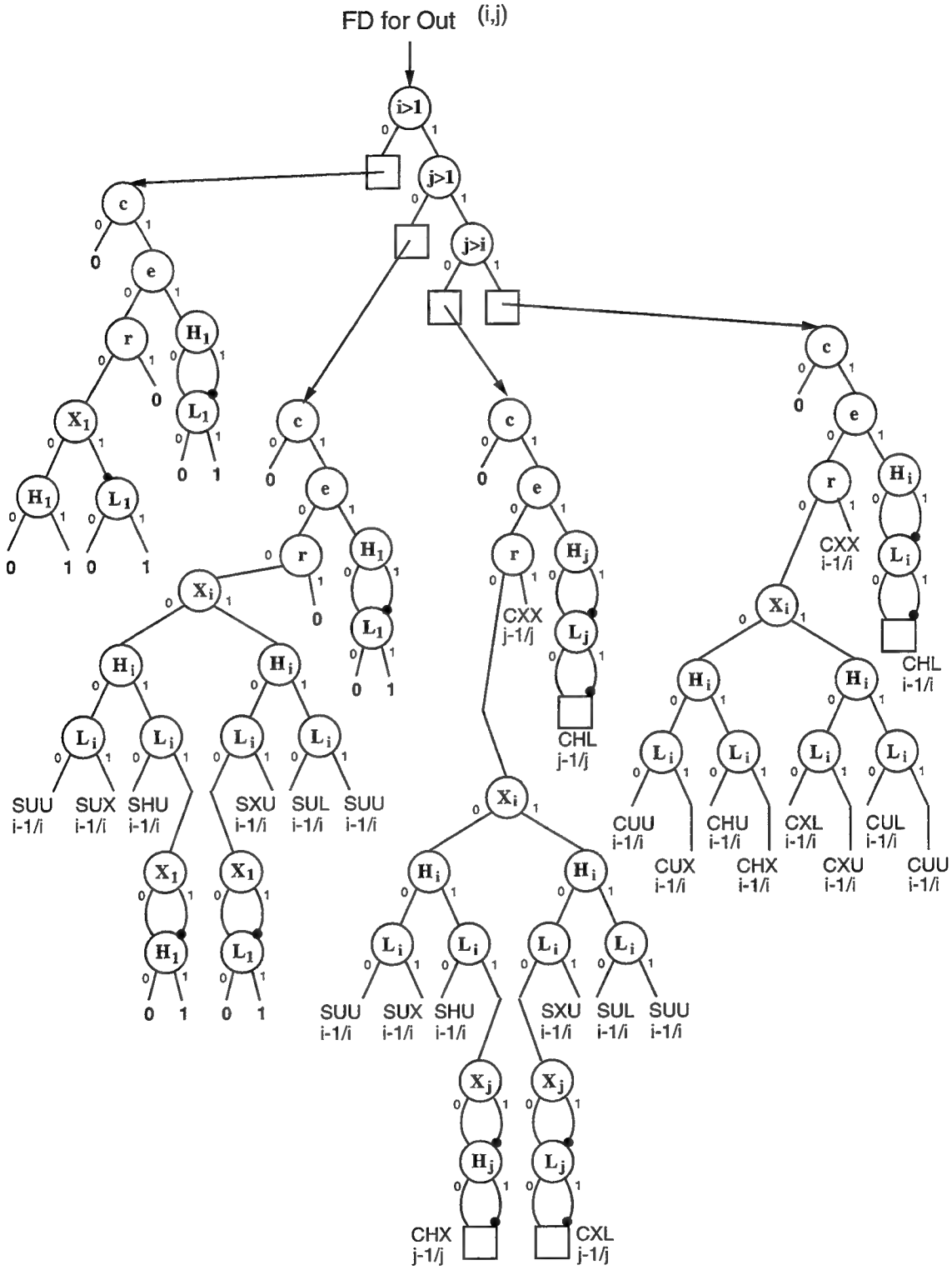


Figure 4.28: LIF Representations for the MINMAX Circuit Outputs

- H – denoting that the argument is known to be the data input H
- L – denoting that the argument is known to be the data input L
- U – denoting that this argument is still unknown

This allows the influence of X_i, H_i and L_i on the j^{th}-bit of the output, as can be clearly seen in the two LIBDDs at the bottom.

4.6 Composition of Inductively-defined Circuits

Supporting a composition of inductively-defined circuits is an important aspect of any methodology dealing with parametric circuits. In the IBF methodology, this support is provided by automatically obtaining representation of a circuit from representations of its components. Since the component descriptions are parametric, this typically involves resolution of various kinds of parameter interactions, summarized as follows:

- Single-instance composition – for a given i, f^i is substituted as an i-instance input x^i in g^i (denoted $g^i|_{x^i=f^i}$). This can be represented by Boolean operations and restriction [22] as:

  $$g^i|_{x^i=f^i} = f^i \wedge g^i|_{x^i=1} \vee \neg f^i \wedge g^i|_{x^i=0}$$

  These operations are easily handled by symbolic LIF manipulation algorithms. Note that the substitution for x^i does not affect any $(i-1)$-instance functions appearing in the definition of g^i.

- All-instance composition with identical parameters – for all $i \geq 1$, f^i is substituted as an i-instance input x^i in g^i, e.g. composition of an i-bit serial parity circuit inputs with outputs from an i-bit serial adder.

  This case is an extension of the previous one, where substitution for all instances is implemented by applying the operation on all $(i-1)$-instance functions also. This has been described in detail in Chapter 3.

- All-instance composition with different parameters – for all $i \geq 1, j \geq 1$, $f^{(i,j)}$ is substituted as an i-instance input x^i in g^i, e.g. the read/write ports of a $(2^i \times j)$-bit register file (Example 17) can be composed with the inputs/outputs, respectively of an i-bit ripple carry adder (Example 1).

  This case is more interesting as it requires introduction of an extra parameter in the composed description for g. It is handled by representing explicit vectors of outputs (described in Section 4.4.3) to capture dependence of the x inputs on the extra parameter.

- Basis instance composition – for $i = 1$, $f(y)$ is substituted as the basis instance input x^1 in g^1, where y represents a set of inputs completely disjoint from the inputs of g, e.g. in a carry lookahead adder, the carry-in for the right half of a carry generate-propagate unit (Example 6) is provided by a function on left half of the inputs.

  Currently, the canonical representation for f is simply substituted in the the Basis BDD for g.

Chapter 5

LIFs and Classic DFAs

As described in the previous chapters, LIFs naturally capture the temporal induction inherent in sequential system descriptions, where functions at time t are defined in terms of inputs at time t and functions at time $(t-1)$. In this sense, state and output functions of a deterministic finite state machine (FSM) can be regarded as LIFs with time as the induction parameter. Thus, the LIF representation directly provides a canonical representation for a sequential function (independent of a particular state encoding), in much the same way as a Binary Decision Diagram (BDD) provides a canonical representation for a combinational function [22].

On the other hand, a minimal *deterministic finite state automaton (DFA)* can also be regarded as a canonical representation of a sequential function, where acceptance/rejection on an input string denotes Boolean value '1'/'0' respectively) [79]. Therefore, a natural question is: *What relationship, if any, is there between the LIF representation and a classic minimal DFA representation of a sequential function?* In this chapter, this question is answered by showing that the LIF representation corresponds to a minimal *reverse* DFA, i.e. a DFA which accepts the input strings in reverse order.

The interesting aspect of this correspondence is that, in some cases, a reverse DFA is exponentially more compact than the forward DFA. Though this observation is easy to understand in theory, yet no effort has been made to examine its potential in practice. As shown in this chapter, several practical datapath circuits such as register files, shift registers, stacks, FIFOs etc. exhibit this exponential compaction when represented as reverse DFAs. Thus, the LIF representations (reverse DFAs) for such circuits offer a distinct advantage over classic (forward) DFA representations. Since the problem of state explosion generally becomes worse in the presence of such datapath circuits, due to the multiplicative effect of individual components, this reduction in state space helps in making the state space exploration task tractable, which is a key component of many formal verification techniques.

While it is useful to understand the conceptual correspondence of the LIF representation to a reverse DFA, there are key differences with respect to a traditional representation of a DFA

as a state transition diagram. Apart from the standard compactness advantages of a symbolic representation, the LIF representations allows simultaneous exploitation of both decomposition and state-sharing naturally. This is in contrast to traditional DFA representations where one can exploit only one or the other, as demonstrated with examples. A summary of these tradeoffs with regard to canonical sequential function representations was presented in a recent paper [70].

This chapter starts by summarizing the symbolic sequential function manipulation framework provided by the LIF schemata. It is intended to supplement the technical details described in Chapter 3 for generic LIFs, by providing specific examples in the context of classic sequential function applications. This also brings out more clearly the relationship between LIF manipulations and standard language/automata-based methods for manipulating sequential functions.

5.1 Symbolic Manipulation of Sequential Functions

Symbolic manipulation of sequential functions is performed by using graph-based algorithms on the corresponding LIF representations, as described in Chapter 3. The feature, that most of these algorithms are based on extensions of BDDs, provides a strong correspondence between symbolic manipulations in the combinational world of Boolean functions and the sequential world of LIFs. Given the practical importance of sequential functions in all aspects of digital system design, it is interesting to study this further.

Table 5.1 summarizes the correspondence between BDD manipulations on one hand, and LIF manipulations on the other. The most important is the fact that just as BDDs capture canonicity of Boolean functions over inputs (with respect to a given variable ordering), the LIF representations capture canonicity of a Boolean function over *strings*[1] of inputs (also, with respect to a given variable ordering). As remarked earlier, each individual i-instance of an LIF can be regarded as a Boolean function of i-length input strings. By capturing all arbitrary (but finite) instances of an LIF in a parametric canonical form, the LIF representation provides canonicity over input strings of all arbitrary (but finite) lengths. Since regular input sequences can also be characterized in the same manner [79], LIF representations capture canonicity over regular input sequences. (Note that this claim is not being made for ω - regular input sequences, which require more complicated acceptance predicates such as in Büchi automata, Muller automata [141].) Thus, LIF representations bridge the gap between combinational and sequential functions, as also between Boolean functions and regular languages.

Since combinational functions are those Boolean functions that depend only on the current set of inputs, canonicity over inputs translates easily to checking combinational function equivalence with BDDs. On the other hand, since sequential functions may depend on the current as well as the past set of inputs (along with an initial state), canonicity over input strings

[1] In general, a string is regarded as a finite-length sequence.

BDD Manipulations	LIF Manipulations
Canonicity over inputs	Canonicity over input strings
Combinational function equivalence	Sequential function equivalence
Combinational don't-care conditions	Sequential don't-care conditions
Boolean function equivalence	Regular language equivalence
Boolean function implication	Regular language containment
Satisfiability	Satisfiability: Reachability
– Function true for some input	– State predicate true for some input string
Tautology	Tautology: Invariant properties
– Function true for all inputs	– State predicate true for all input strings

Table 5.1: Correspondence of BDD and LIF Manipulations

(combined with a basis function) allows checking of sequential function equivalence with LIF representations. Traditionally, the field of combinational circuits has been much better understood, with many combinational synthesis and analysis techniques being extended to the field of sequential circuits. One of these is the use of "don't-cares" to capture unknown or irrelevant behavior [20]. Just as BDDs allow easy symbolic manipulation of the don't-care input conditions for combinational circuits, LIF representations allow the same for sequential circuits, where don't-cares are represented in the form of a set of input strings (and initial state). These can then be algebraically manipulated in the same manner as other LIFs in the system. Furthermore, some special symbolic operations found to be practically useful with BDDs in order to handle don't-cares, such as the *constraint operator* [48], the *generalized co-factor operator* [142] etc., can apply directly to LIF manipulations also. An interesting application of this is in minimization of incompletely specified finite state machines, as described in more detail later in this section.

The correspondence between BDDs and LIFs is also very useful for treating regular languages in a manner similar to Boolean functions. Note that a regular language can be denoted by a sequential function, i.e. a predicate on state and input variables in the form of an LIF, which uses some state-encoding. However, since LIF representations are independent of the state-encoding, they are useful for handling regular languages directly, without getting caught in the merits and demerits of a particular state-encoding. Thus, Boolean function equivalence with BDDs corresponds to regular language equivalence with LIFs (similar to combinational and sequential function equivalence, respectively). Furthermore, the Boolean function implication operation with BDDs corresponds to regular language containment with LIFs, i.e. for LIFs A and B, $(A \Rightarrow B)$ signifies language containment because every input string accepted by A is also accepted by B. Satisfiability, i.e. truth of the Boolean function for some assignment

of inputs in the BDD world, corresponds to reachability in the LIF world, i.e. truth of the state predicate for some assignment of the input string. Similarly, a tautology, i.e. truth of the Boolean function for all assignments of inputs with BDDs, corresponds to invariant properties with LIFs, i.e. truth of the state predicate for all assignments of input strings.

The advantages of treating regular languages and finite state sequential systems in the form of Boolean functions are manifold. The emphasis on the algebraic framework allows the operational details based on automata semantics to be abstracted out, i.e. there is no need to always view finite state machines in terms of states and transitions; for some applications the underlying sequential functions may suffice. Furthermore, it allows the entire spectrum of well-developed BDD techniques to be applied in the context of finite state systems. In the following subsections, some of these are illustrated with reference classic finite state machine problems.

5.1.1 Reachable States of an FSM

Consider the following problem, described by Coudert, Berthet and Madre [47, 48]:

Given a Boolean function $F : B^m \rightarrow B^n$, represented as a vector of Boolean functions $F = [f_1, f_2, \ldots f_n]$, where each f_i is defined over m symbolic variables $X = [x_1, x_2, \ldots, x_m]$, obtain a symbolic representation of the range of F.

Definition 7: The range of F can be represented by its characteristic function, denoted $\Phi(F)$, defined as:

$$\Phi(F)(y_1, y_2, \ldots y_n) = \exists x_1 . \exists x_2 . \ldots \exists x_m . ((y_1 = f_1) \wedge (y_2 = f_2) \wedge \ldots \wedge (y_n = f_n))$$

Note that the function $\Phi(F)$ is symbolically represented in terms of independent variables $Y = [y_1, y_2, \ldots, y_n]$. It can be obtained by using standard BDD techniques for the existential quantification and Boolean operations in the above equation. Some performance enhancements have also been shown successful, which are especially effective in the context of the inner-loop computation for checking input/output equivalence of deterministic FSMs [48].

An interesting application of this technique in the LIF framework is to obtain the symbolic representation of the reachable states of an FSM. Note that for the symbolic state vector $S = [s_1, s_2, \ldots, s_n]$, where each s_i is a state transition function represented as an LIF in terms of a time parameter, $\Phi(S)$ represents the range of S for all time instants, i.e. the set of all reachable states. Indeed, $\Phi(S)$ can be obtained in the same fashion as with BDDs, except that the variables X denote parameterized inputs (in terms of the time parameter), and the corresponding LIF operations are used for the existential quantification and the Boolean operations.

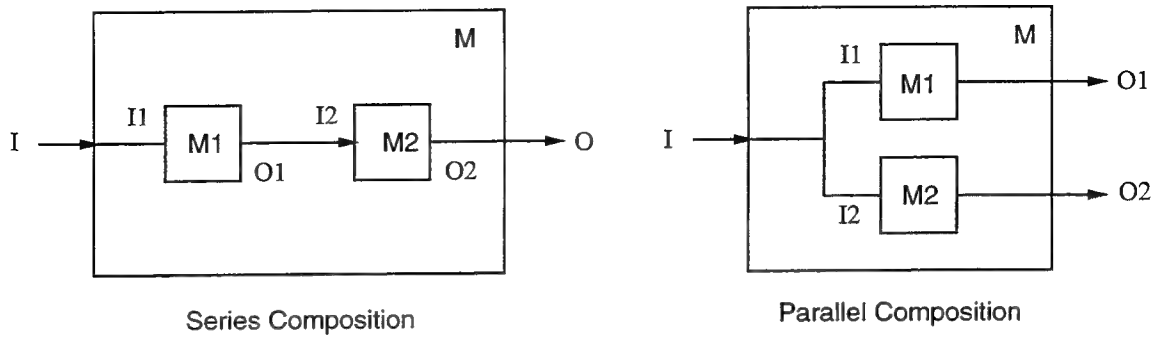


Figure 5.1: Composition of FSMs

5.1.2 Composition of FSMs

Handling composition of FSMs is an important practical problem, since most sequential systems are designed by composing together individually-designed components. This allows a modular approach to system design, as well as reuse of existing components.

Composition of FSMs can either be *synchronous* or *asynchronous*, depending on whether or not they change state simultaneously. Technically, this synchrony refers only to the aspect of composition, not to the individual FSMs itself, i.e. it is possible to define an asynchronous composition of synchronous FSMs. In practice, the simplest form is a synchronous composition of synchronous FSMs. This has been studied in the following cases, also shown is Figure 5.1 for two machines $M1$ and $M2$ (extendible to n machines):

- **Serial Composition**: where the outputs of $M1$ feed into the inputs of $M2$

  Definition 8: Given two FSMs[2]

  $M1 = (Q1, \Sigma, \Delta1, T1, O1, q1_0)$,
  $M2 = (Q2, \Delta1, \Delta2, T2, O2, q2_0)$,

  the serial composition machine $M = M1 \times_s M2 = (Q, \Sigma, \Delta2, T, O, q_0)$ is defined such that:

  - $Q = Q1 \times Q2$
  - $T((a1, a2), \sigma) = (b1, b2)$,
    where $T1(a1, \sigma) = b1$, $O1(a1, \sigma) = \delta1$ and $T2(a2, \delta1) = b2$
  - $O((a1, a2), \sigma) = \delta2$,
    where $O1(a1, \sigma) = \delta1$ and $O2(a2, \delta1) = \delta2$

[2]The Mealy machine model used here has been defined in Chapter 2, Definition 3.

- $q_0 = (q1_0, q2_0)$

- **Parallel Composition**: where both $M1$ and $M2$ operate independently on the common set of primary inputs

  Definition 9: Given two FSMs
  $M1 = (Q1, \Sigma, \Delta1, T1, O1, q1_0)$,
  $M2 = (Q2, \Sigma, \Delta2, T2, O2, q2_0)$,
  the parallel composition machine $M = M1 \times_p M2 = (Q, \Sigma, \Delta, T, O, q_0)$ is defined such that:

  - $Q = Q1 \times Q2$
  - $\Delta = \Delta1 \times \Delta2$
  - $T((a1, a2), \sigma) = (b1, b2)$,
    where $T1(a1, \sigma) = b1$ and $T2(a2, \sigma) = b2$
  - $O((a1, a2), \sigma) = (\delta1, \delta2)$,
    where $O1(a1, \sigma) = \delta1$ and $O2(a2, \sigma) = \delta2$
  - $q_0 = (q1_0, q2_0)$

For serial composition, the composition of outputs of $M1$ with inputs of $M2$ can be handled by using the all-instance LIF composition (described in Section 3.2.3, Chapter 3). It can be illustrated with following simple example:

Example 19: Consider the composition of two 1-bit counters to form a 2-bit counter as shown in Figure 5.2. The enable input $e2_t$ of the second counter is composed with $f^t = e1_t \wedge s1^{t-1}$, $t > 1$, where $e1$ and $s1$ are the enable input and the state of the first counter, respectively. The LIF representations of the original 1-bit counter states $s1, s2$, and the function f are as shown in Figure 5.3 Part(a). The LIF representations of the composed circuit can be easily derived from these, by substitution of f for $e2$ in the representation of $s2$ as follows:

$s2|_{e2=f} = f \wedge s2|_{e2=1} \vee \neg f \wedge s2|_{e2=0}$

These operations lead to the final representation as shown in Figure 5.3 Part(b). Note that this is identical to the LIF representation of a standard 2-bit counter shown earlier in Figure 4.4. This also confirms automatically that the given composition of two 1-bit counters is indeed equivalent to a 2-bit counter.

Parallel compositions of FSMs are easier to handle in practice, since they essentially reflect an independent partition of the combined state space. Within the LIF framework, since each FSM output is represented as an independent LIF, its LIF representation is unaffected by any parallel composition operation. As for the state-space representation, each of the state transition

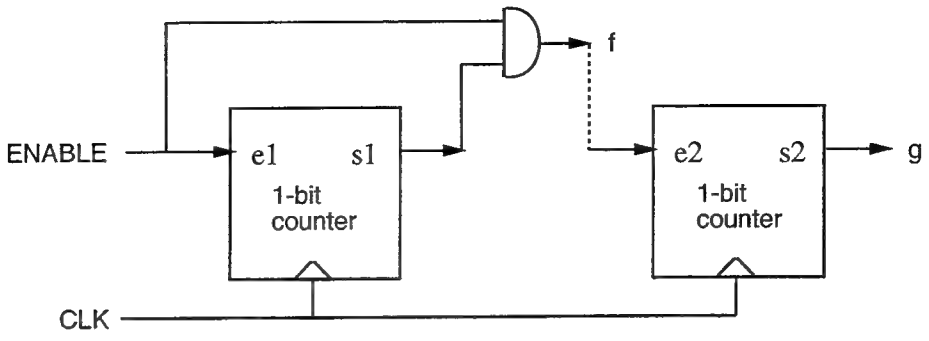


Figure 5.2: Composition of Two 1-bit Counters

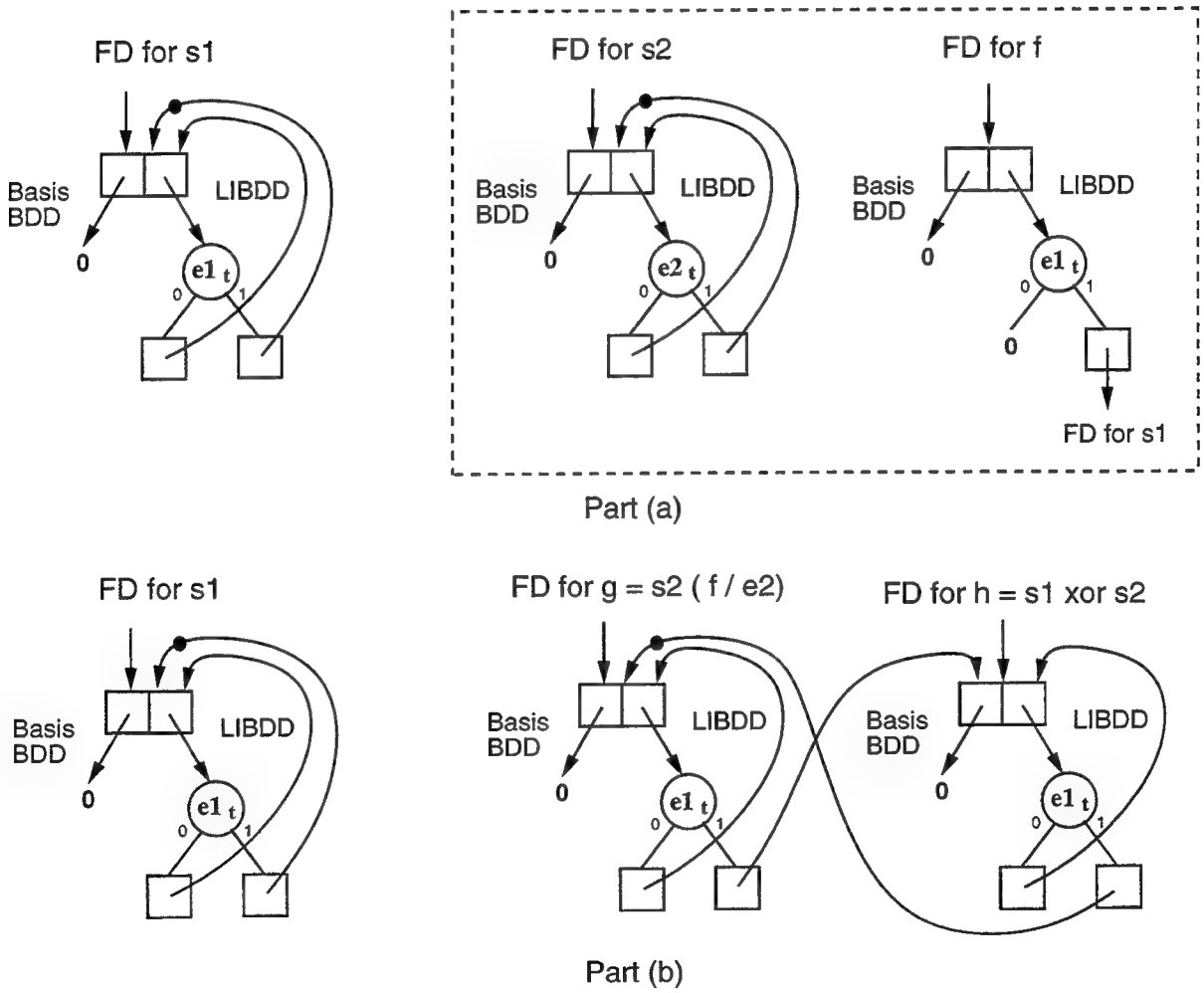


Figure 5.3: LIF representation for Composition of Two 1-bit Counters

LIFs also remain unaffected. The only change is in the potential representation of the set of reachable states using the technique outlined in the previous section. The combined symbolic state vector $S = [S1, S2]$ needs to be considered, where $S1$ and $S2$ are the symbolic state vectors of machines $M1$ and $M2$, respectively. The set of reachable states of the product machine is $\Phi(S)$.

5.1.3 Handling Sequential Don't-Cares

As briefly mentioned earlier, "don't-cares" are often used to capture unknown or irrelevant behavior [20]. For example, two combinational functions a and b may not be equivalent with respect to all inputs. However if it is known that input combinations characterized by a don't-care function d are irrelevant, then for all practical purposes the two can be regarded as equivalent if the formula $\neg(a \oplus b) \vee d$ is a tautology. Furthermore, even if a and b are equivalent for all inputs, it may be easier to check the above formula than to check $\neg(a \oplus b)$, due to the potential simplification in the BDD representation.

Similarly, in the case of two sequential functions f and g, if the don't-care input sequences can be characterized by an LIF d_{seq}, then f and g can be regarded equivalent if the LIF formula $\neg(f \oplus g) \vee d_{seq}$ is a tautology (i.e. its value is 1 at all time instants $t \geq 1$). From the verification viewpoint, this is a significant advantage in practice, since there may be certain input sequences which never arise in a system, even though each of the individual inputs are valid inputs. In such cases, it is not enough to use a combinational analysis of the input combinations to exploit the don't-cares. Rather, a sequential analysis capability is needed to deal directly with don't-care sequences. Since regular don't-care sequences can be captured as LIFs, this capability is provided very naturally with LIF manipulation in in these cases. Note that such direct manipulation with don't cares subsumes a simpler analysis based on the characterization of reachable states, or removal of redundant states by minimization [100]. Furthermore, many special BDD techniques proposed for the exploitation of combinational don't-cares, such as the *constraint operator* [48], the *generalized co-factor operator* [142] etc., can be directly used with the LIF representations also, in order to simplify sequential function manipulation.

5.1.4 Handling Incompletely-Specified FSMs

An interesting application of exploiting sequential don't-cares can be seen in handling input incompletely-specified machines, which has recently been the subject of renewed interest [51, 71, 129, 152]. The problem can be described as follows [152]): consider a cascade connection of two machines $M1$ and $M2$, as shown in Figure 5.4. Unger observed that when driven by $M1$, the machine $M2$ may contain more unspecified transitions than as an isolated machine, and described a method to simplify $M2$ based on exploiting some of these transitions [122, 143].

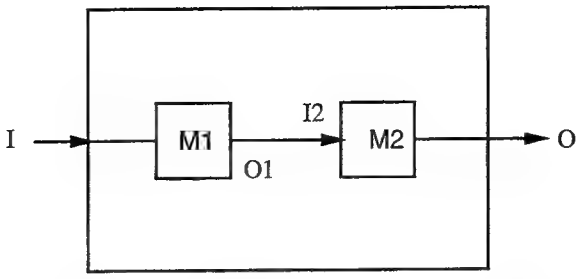


Figure 5.4: Cascade Connection of FSMs

A complete solution was provided by Kim and Newborn [92], which can be summarized as follows:

- Obtain a nondeterministic finite state automaton (NFA) $\mathcal{A}$ which accepts the output strings *produced* by the machine $M1$, where a dummy *dead* state is introduced to catch transitions on all unspecified output symbols.
- By using subset construction [128] and state minimization [79] on $\mathcal{A}$, obtain the equivalent, minimal, deterministic finite state automaton (DFA) $\mathcal{A}'$ corresponding to $\mathcal{A}$.
- Compose $\mathcal{A}'$ with $M2$, and delete any transitions to the *dead* state leading to a simplified machine $M2'$.

Note that $\mathcal{A}'$ captures the output language of the machine $M1$. Thus, the output sequences that are never produced by $M1$ serve as the input don't-care sequences for $M2$, which are then converted to unspecified transitions of the simplified machine $M2'$. The difficult part of this procedure is the subset construction involved in the second step, which is of exponential complexity. Various techniques have been proposed to handle it in practice, along with extensions that allow handling of arbitrary network connections [129, 152].

LIF manipulations offer yet another alternative within this spectrum, with the interesting feature that *the LIF representation for DFA $\mathcal{A}'$ is obtained directly from $M1$, without going through the steps of obtaining the NFA $\mathcal{A}$ and subset construction.* Since this task is also of exponential complexity, no bounds are being violated here. However, given that the LIF technique for obtaining $\mathcal{A}'$ is different from those studied so far, there may be practical advantages to using it. This technique for representing the output language of an FSM is described in detail in the next section, and its relationship to standard DFA techniques is described later in the chapter. Once the LIF representation for $\mathcal{A}'$ is obtained, it can be used in the standard way to minimize the machine $M2$. Either direct composition recommended by Kim and Newborn can be used (described in Section 5.1.2), or it can be used with a constraint operator [48]/generalized co-factor operator [142] in order to simplify the LIF representations of the outputs and state-transition LIFs of $M2$ (with respect to a fixed variable ordering).

5.1.5 Representation of the Output Language of an FSM

Consider an FSM M with output vector $O = [o_1, o_2, \ldots o_k]$, state vector $S = [s_1, s_2, \ldots s_n]$, and input vector $X = [x_1, x_2, \ldots x_m]$. Define a new parametric function $\Gamma^t(O)$ as follows:

Definition 10: for $t \geq 1$, $\Gamma^t(O) = \Phi([O^1, O^2, O^3, \ldots O^t])$

where O^t denotes the value of the output vector at time instant t,
and $\Phi(F)$ denotes the characteristic function of the range of a Boolean function vector $F = [f_1, f_2, \ldots f_t]$, $f_j : B^m \rightarrow B$ [48] (described in detail in Chapter 5, Section 5.1.1), defined as follows:

$$\Phi(F)(v_1, v_2, \ldots v_t) = \exists w_1 . \exists w_2 . \ldots \exists w_m . ((v_1 = f_1) \wedge (v_2 = f_2) \wedge \ldots \wedge (v_t = f_t))$$

Note that the above definition of $\Phi(F)$ can also apply to vectors of vectors of Boolean functions, e.g. in the definition of $\Gamma^t(O)$. In this case, the right-hand side of Definition 10 can be represented as a Boolean formula on variables $[Y^1, Y^2, \ldots Y^t]$, where each (vector) $Y^j = [y_1^j, y_2^j, \ldots y_k^j]$ is introduced for each (vector) $O^j = [o_1^j, o_2^j, \ldots o_k^j]$, $1 \leq j \leq t$, respectively. Thus, $\Gamma^t(O)$ is a function of $[Y^1, Y^2, \ldots Y^t]$, i.e. parameterized variables $y_1, y_2, \ldots y_k$ (parametric in t).

Note also that due to the definition of $\Phi(.)$, $\Gamma^t(O)$ corresponds to the language of all t-length strings *produced* by M. Thus, the output language of M is captured parametrically by $\Gamma^t(O), t \geq 1$. Furthermore, since the output language of an FSM is also regular [79], $\Gamma(O)$ is indeed an LIF with parameter t and parameterized variables $y_1, y_2, \ldots y_k$.

The LIF representation of $f = \Gamma(O)$ can be obtained by the following procedure:

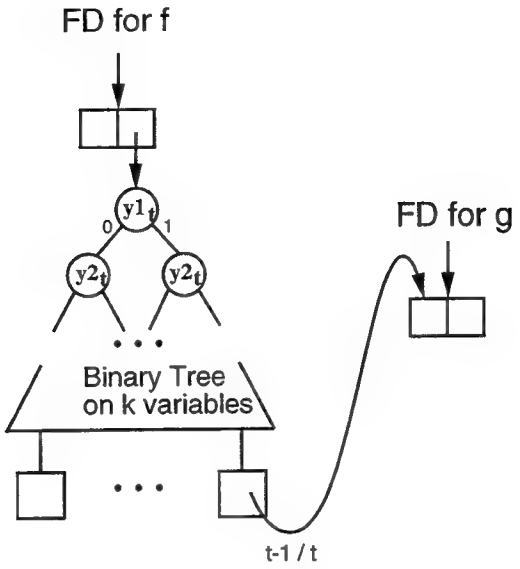
1. Consider an LIF-version of the $\Phi(.)$ operation, called $\Phi_{lif}(.)$, which operates on a single level of the LIBDD at a time as follows:

   $$\begin{aligned} & f^t([y_1^t, y_2^t, \ldots, y_k^t], \ldots) \\ & \quad = \Phi_{lif}(O^t = [o_1^t, o_2^t, \ldots o_k^t]) \\ & \quad = \exists X^t . ((y_1^t = o_1^t) \wedge (y_2^t = o_2^t) \wedge \ldots \wedge (y_k^t = o_k^t)), \end{aligned}$$

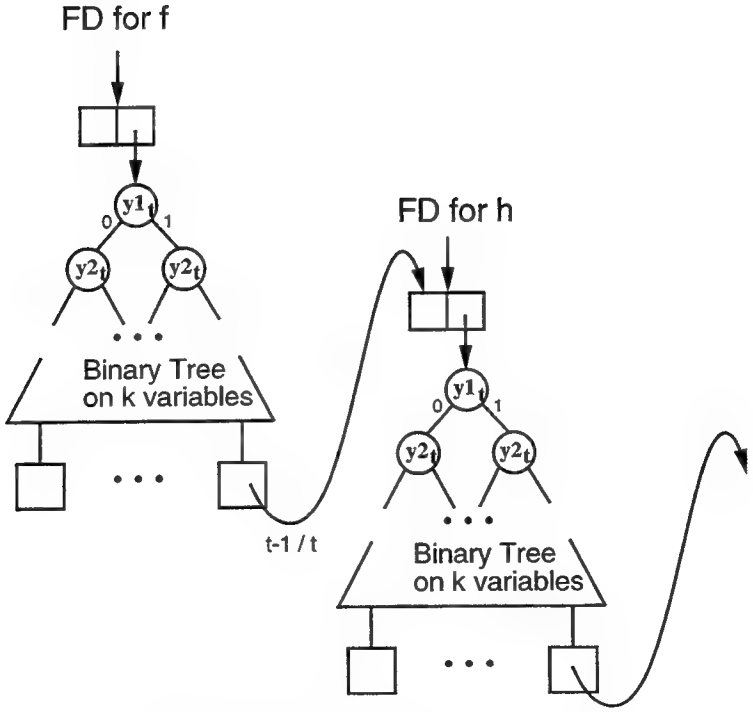
   where $\exists X^t$ is a shorthand notation for $\exists x_1^t . \exists x_2^t . \ldots . \exists x_m^t$.

   Thus, at level t of the Φ_{lif} operation, as shown schematically in Figure 5.5 Part(a):

   - new variables y_j^t are introduced for outputs o_j^t, ($1 \leq j \leq k$, respectively) in the form of a complete binary tree (before reduction),
   - t-instance inputs to M, X^t, are existentially quantified,



Part (a) : After Step 1



Part (b) : After Step 2

Figure 5.5: LIF Representation for the Output Language of an FSM

- resulting in an LIBDD where terminal nodes may contain pointers to FDs with parameter $(t-1)$, e.g. $g^{(t-1)}$.

In some sense, this step amounts to obtaining the characteristic function of the output values at time instant t, where the constraints imposed on states at time instant $(t-1)$ are captured by terminal nodes of the LIBDD.

2. For each terminal node in the resulting LIBDD with a pointer to an FD, say $g^{(t-1)}$, define a new LIF h using the operation Φ_{lif} as follows:

$$\begin{aligned} & h^{(t-1)}([y_1^{(t-1)}, y_2^{(t-1)}, \ldots, y_k^{(t-1)}], \ldots) \\ & \quad = \Phi_{lif}(O^{(t-1)} \wedge g^{(t-1)}) \\ & \quad = \exists X^{(t-1)}.((y_1^{(t-1)} = o_1^{(t-1)}) \wedge (y_2^{(t-1)} = o_2^{(t-1)}) \wedge \ldots \wedge (y_k^{(t-1)} = o_k^{(t-1)}) \wedge g^{(t-1)}) \end{aligned}$$

Therefore, at level $(t-1)$:

- new variables $y_i^{(t-1)}$ are introduced for the outputs $o_i^{(t-1)}$ $(1 \leq i \leq k)$,
- a conjunction is performed with $g^{(t-1)}$ (expressed in terms of inputs $X^{(t-1)}$ and pointers to FDs with parameter $(t-2)$)
- $(t-1)$-instance inputs to M, $X^{(t-1)}$, are existentially quantified,
- resulting in an LIBDD where terminal nodes may contain pointers to FDs with parameter $(t-2)$.

Note that the combined Steps 1 and 2 amount to obtaining the characteristic function for output sequences at time instants t and $(t-1)$, expressed in terms of constraints imposed on states at time instant $(t-2)$, *while* taking into account the constraints generated at the end of Step 1. However, rather than a monolithic step for this operation, new LIFs are introduced for time instant $(t-1)$, in order to exploit the LIBDD representation which is parametric in t. This is shown schematically in Figure 5.5 Part(b). (Recall that each $(t-1)$-instance LIF is implicitly represented as a t-instance LIF with parameter substitution. The Basis BDD for each new LIF is obtained by considering the case $t=1$ in its definition.)

3. Repeat Step 2 for each terminal node in the LIBDD for $h^{(t-1)}$, possibly leading to new LIFs. Repeat it recursively till no new LIFs are needed. Since the number of output LIFs and state-transition LIFs is finite, this procedure is guaranteed to terminate.

The final LIF representation for f characterizes the output language of M in terms of parameterized "output" variables $y_1, y_2, \ldots, y_k$. This technique is also very useful in the context of obtaining automatic network invariants, which utilizes language containment between the languages produced by networks of different lengths. This is described in more detail in Chapter 6 which presents the verification applications of the IBF methodology.

5.1.6 Handling Nondeterminism

One of the limitations of the LIF framework, as described so far, is that it applies directly to only *deterministic* finite state sequential systems. This follows from the fact that only state-transition functions, not state-transition relations, can be captured as LIFs. It is well-known that nondeterminism is a very useful abstraction which serves multiple purposes, e.g. hiding of unnecessary design detail, modeling a lack of information, modeling an unconstrained environment etc. Thus, it greatly helps in controlling complexity at higher levels of the hardware abstraction hierarchy, from the viewpoints of both representation and verification.

A possible technique for handling nondeterminism within the LIF framework is by the introduction of "choice variables" to model nondeterministic choice between a set of deterministic actions. For example, a nondeterministic transition to one of 8 possible successor states can be modeled by the introduction of 3 choice variables in the state-transition LIFs. Note also that n choice variables can model a 2^n-way choice at *each* state in the system. Since practical system descriptions do not exhibit a very large number of choices per state, this technique may work well in practice.

In a sense, this is no different than *input nondeterminism* [39], i.e. the branching caused by system inputs in the transition structure of a model (such as a Kripke structure [41]). Furthermore, in most verification applications – e.g. model checking, FSM equivalence checking – either an existential or a universal quantification on all transitions is required. Both of these are easily implemented as standard Boolean operations for quantification on the choice variables. This technique can also be used to capture a set of initial states for finite state machines/automata, which can then be appropriately manipulated according to the desired operation. In this regard, it is also interesting to note that in many applications involving transition relations, a candidate is chosen deterministically from each class (according to a fixed criterion), as a representative for all subsequent manipulation [99, 152].

5.2 Relationship between LIF Representations and DFAs

A deterministic finite state automaton (DFA) is similar to a finite state machine, except that it does not have an output alphabet. Instead, it has a designated set of *final states*. An input string is *accepted* or *rejected*, depending on whether the automaton is in a final state or not, respectively. In formal terms:

Definition 11: A deterministic finite state automaton is a 5-tuple (Q, Σ, T, q_0, F), where

- Q is a finite set of states
- Σ is an input alphabet

- T is a transition function, $T : Q \times \Sigma \rightarrow Q$
- q_0 is the initial state, $q_0 \in Q$
- F is a set of final states, $F \subset Q$

The language accepted by a DFA, i.e. the set of all input strings $\sigma \in \Sigma^*$ accepted by a DFA, is regular; conversely, any regular language is accepted by some DFA [79]. Let $DFA(\mathcal{L})$ denote a DFA that accepts a regular language $\mathcal{L}$. A regular language $\mathcal{L}$ also corresponds to a sequential function L, where $L(\sigma) = 1$ if $\sigma \in \mathcal{L}$, and $L(\sigma) = 0$ if $\sigma \in \mathcal{L}$. Typically, L is implemented by a sequential circuit which can use any state encoding.

Definition 12: For a language $\mathcal{L}$, define $\mathcal{L}^R$ to be the *reverse language*, i.e. $\mathcal{L}^R = \{x^R | x \in \mathcal{L}\}$, where x^R denotes the reverse of string x.

For example, consider a regular language $\mathcal{L} = (0+1)^*1(0+1)^k$, which corresponds to all $0/1$ strings where the last $(k+1)^{th}$ input is a 1. Its reverse language $\mathcal{L}^R = (0+1)^k1(0+1)^*$, i.e. all $0/1$ strings where the *first* $(k+1)^{th}$ input is a 1. If $\mathcal{L}$ is regular, then so is $\mathcal{L}^R$. Therefore, there exists a DFA which accepts $\mathcal{L}^R$, denoted by $DFA(\mathcal{L}^R)$. Such a DFA can be considered a *Reverse DFA* for $\mathcal{L}$.

The relationship between LIF representations and DFAs can now be stated as follows:

Theorem 3: *LIF representation of* $L \equiv$ *minimal* $DFA(\mathcal{L}^R)$.

The proof follows from the simple observation that a traversal originating from the LIF representation for L, is equivalent to following a transition path through a DFA, where:

- FDs are regarded as states,
- parameterized variables of the LIBDDs are regarded as inputs, and decision paths through LIBDDs are regarded as input labels,
- pointers from LIBDD terminal nodes to FDs are regarded as transitions to new states,
- the FD for L is regarded as the initial state, and
- the Basis BDD of the last FD in the traversal is regarded as the final acceptance condition.

Note that for a finite number of user-given LIFs, i.e. a finite number of transition functions, there can be only a finite number of FDs, i.e. states. Furthermore, since the LIF representation for L is independent of the state encoding, it justifies the arbitrary state encoding assumption for L.

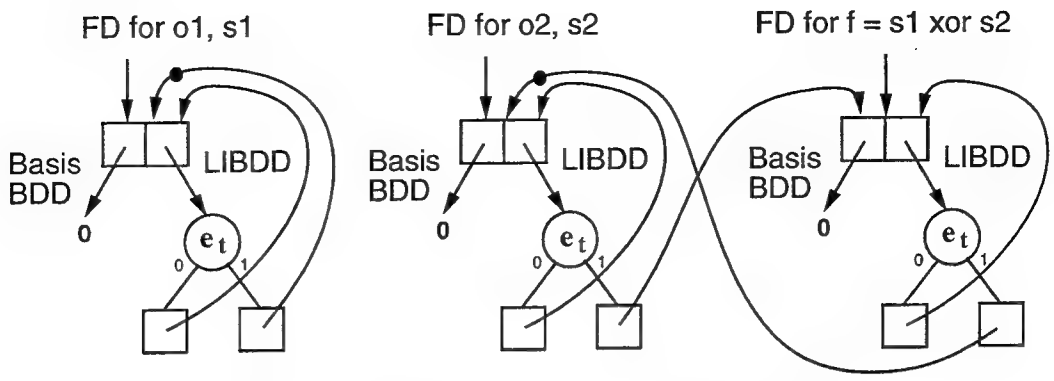


Figure 5.6: LIF Representation for a 2-bit Counter

As a concrete example of such a traversal, consider the 2-bit counter example again, reproduced here in Figure 5.6. Note that the LIBDD for the output $o2$ represents its t-instance inductively. Thus, for a given input string $x = x^1x^2 \ldots x^t$, the value of o^t (the t-instance of $o2$) is obtained by traversing the LIBDD, branching on the input value $e^t = x^t$, and reaching the associated LIBDD terminal node. Suppose $x^t = 1$, and we reach the terminal node on the right which contains a pointer to the FD for f. Recall also that this pointer implicitly represents a parameter substitution $(t-1)/t$. In other words, it represents $f^{(t-1)}$, the $(t-1)$-instance of f. Again, the value of $f^{(t-1)}$ is obtained by traversing its LIBDD, branching on the input value $e^{(t-1)} = x^{(t-1)}$, and following its terminal node pointer. This process is repeated, until we require g^1 for some LIF g. This is obtained by following the Basis BDD in the FD for g, branching on the basis input $e^1 = x^1$, leading finally to a Boolean constant. This Boolean constant denotes the value of the output $o2^t$ for the input string x, where a '1' indicates acceptance, and a '0' indicates rejection by the DFA.

Thus, a traversal of the LIF representation is like a DFA traversal. However, note from the example that we have consumed the last input first when starting from the FD for L, i.e. the input string is followed in the reverse order $x^t x^{(t-1)} \ldots x^2 x^1$ for the traversal. In other words, the LIF representation for L corresponds to a DFA which accepts the reverse of input strings contained in $\mathcal{L}$, i.e. the LIF representation for $L \equiv DFA(\mathcal{L}^R)$. As for minimality of $DFA(\mathcal{L}^R)$, recall that the LIF canonicity algorithm ensures the pairwise non-equivalence of all FDs with respect to input strings. Therefore, the states of the corresponding reverse DFA are unique. Hence, LIF representation for $L \equiv$ minimal $DFA(\mathcal{L}^R)$.

Though a traversal of the LIF representation is similar to a traversal of the reverse DFA, there are some crucial differences in comparison to a state transition graph representation of a DFA. First, note that the value of the output o^t is available only at the end of the complete traversal with the (reversed) input string. Unlike the standard DFA traversal, the entire sequence of outputs corresponding to a traversal is not available. If the latter is desired, the traversal has

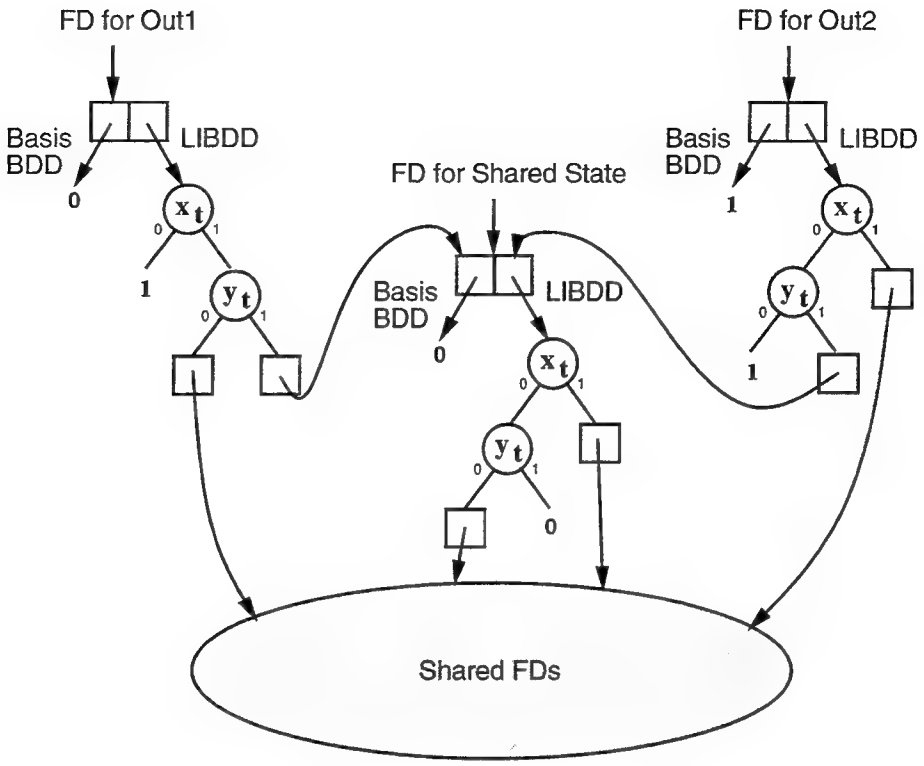


Figure 5.7: Multiroot LIF representations for Sequential Functions

to repeated for every prefix of the input string. This is an artifact of the reverse nature of the LIF representation with respect to the input string. Second, the LIF representations for all sequential functions in the system resemble a *multi-rooted* BDD representation [22], in that each root into an FD node identifies a unique sequential function and all common FDs are shared. This is shown schematically in Figure 5.7. Note that apart from providing a separate handle for each sequential function, this scheme also allows sharing of common states (FDs). In contrast, there is no emphasis on the representation of an individual output in a standard state transition graph, unless it is minimized with respect to the particular output. If the latter is done, the information regarding common states (with other outputs) is lost. This feature of simultaneously providing decomposition (per sequential function) and state-sharing is described in more detail in Section 5.4.

It is also instructive to note that while the LIF representation corresponds to an *explicit* reverse DFA, not all applications require a construction of this DFA. For example, if checking LIF equivalence is required, a breadth-first version of the LIF equivalence algorithm (Figure 3.9, Chapter 3) is equivalent to an *implicit* symbolic backward traversal on the underlying state space. Its relationship to known techniques is described in detail in the next chapter.

5.3 Related Work

5.3.1 Classic DFA Reversal

The concept of reverse DFAs is not new, and their study has been the focus of several early as well as recent works [19, 79, 130, 146]. The primary difference between them and the LIF work described in this thesis is the nature of the method used for obtaining the reverse DFA. The classic method to obtain a reverse DFA is to start from an explicit forward DFA, exchange the initial and final states, and reverse the direction of each transition. Note that this may result in a nondeterministic finite state automaton (NFA). For example, two distinct incoming transitions into a state on the same input, will become two outgoing transitions on the same input upon reversal, thereby requiring nondeterminism. The resulting NFA can then be made deterministic by the standard Scott-Rabin subset construction [128].

In particular, this is the approach followed by Rho and Somenzi [130], who have used a reverse DFA to obtain linear-sized BDD representations for the outputs of a linear iterative array. They start with an explicit forward DFA which captures all combinations of outputs from the iterative array cell, and obtain the reverse DFA by the process described above. Though their target is also inductive verification, their work makes no mention of either the potential use of reverse automata for canonical representation and symbolic manipulation of iterative functions, or its compactness advantage in practice.

On the other hand, recall that the method for obtaining canonical LIF representations (reverse DFAs) starts from standard state transition equations for the sequential system. The LIF representations are obtained in a *lazy* way by unrolling the state transition equations until the required extent, through the process of LIF generation. These representations are subsequently compared for ensuring minimality. Thus, the LIF-based method can be regarded as an *implicit* method for obtaining the minimal reverse DFA, in that it directly uses state transition equations, and does not rely on a representation of the forward DFA at all. It completely avoids the problem of subset construction, which is of exponential complexity in the size of the forward DFA. Though the final reverse DFA is itself explicit, at least it helps to avoid the explicit forward DFA, especially in systems where the reverse DFA is known to be more compact (described later in this chapter). Along the same lines, recall that the LIF technique for obtaining the representation of the output language of an FSM (described in Section 5.1.5) also unrolls the output LIF representations in reverse order along the time parameter, directly obtaining the reverse DFA representation of the output language.

Finally, though the single parameter LIF representation can be naturally viewed in terms of a reverse DFA, it should be noted that this representation forms part of a general representation for multiple parameter LIFs (IBFs) as described in the previous chapter. On one hand, this representation is geared towards handling general parameterization issues (e.g. multiple parameters), which do not fit well within an automata framework. On the other hand, the LIF

representation can be regarded as a functional representation, without the operational semantics associated with a DFA. This is similar to the correspondence between BDDs and read-once two-way automata, noted by several researchers [14, 46, 62]. Though all BDD operations – conjunction, disjunction, quantification etc. – can be viewed equivalently in terms of the automata operations, most applications have successfully exploited these for effective symbolic Boolean manipulation without recourse to the automata semantics. Similarly, the advantages of LIF representations potentially lie in exploiting the effective representations, rather than the underlying semantics, for efficient symbolic sequential function manipulation.

5.3.2 DFA State Minimization

In the reverse DFA view of an LIF representation, each FD corresponds to an explicit state. In fact, it is this explicit nature of the final LIF representation that makes it independent of the state encoding used in the user-provided state transition functions. Furthermore, the canonicity property of the LIF representations directly ensures distinctness of the states (minimality) of the reverse DFA. Exploring this correspondence further, the Comparison Phase of the LIF canonicity algorithm[3] can also be regarded as a state minimization procedure, where each FD is checked for equality against members of the canonical set. However, the overall LIF canonicity algorithm is different in several respects from the standard DFA state minimization algorithms, both with explicit state-space manipulation [2, 78, 79], as well as with implicit (symbolic) state-space manipulation [100, 124].

First, since the LIF canonicity algorithm starts from state transition equations, there is no need for handling the entire unminimized state space. The Generation Phase of the LIF algorithm, which has no counterpart in standard DFA algorithms, lazily generates only as many explicit LIFs (states) as are needed for a particular output. Furthermore, the technique of capturing canonicity with respect to user-defined LIFs (state variables) is new, and has not been exploited even by other symbolic methods. It is this technique which allows the LIF generation process to terminate, while making a conservative estimate of the final equivalence classes. As for the Comparison Phase, it is similar to the explicit state minimization process, where explicit equality checks between pairs of LIFs are performed. However, the complexity of this phase is $O(n^2)$, where n is the number of LIFs at the end of the Generation Phase, *not* the number of unminimized states.

Second, the LIF canonicity algorithm is incremental in nature, relying on reuse of previous results, and allowing interleaving of the Generation and Comparison phases. This is consistent with the aim of supporting symbolic manipulation of a dynamically growing set of sequential functions, where more functions are added as more tasks are performed. Again, the standard algorithms, as stated, are useful for one-time minimization and not for incremental operation.

[3] described in detail in Chapter 3, Section 3.2.2.2

It is also interesting to note that, as stated, the LIF canonicity algorithm obtains equivalence classes taking into account the initial state (at time instant $t = 1$), as captured by the Basis BDDs. In other words, the equivalence check on Basis BDDs (Step 4, Figure 3.9, Chapter 3) ensures the equivalence of outputs on input strings starting from the initial state. However, it is easy to modify this algorithm for a more general scheme of checking equivalence of outputs on input strings starting from any *arbitrary* state, such as that described by Pixley [124]. Essentially, the check on Basis BDDs is omitted, only the equivalence check on LIBDDs is conducted (Step 5, Figure 3.9, Chapter 3), where the recursive checks leading to cycles are resolved by using the principle of induction (Theorem 2, Chapter 3). Furthermore, the set of all *alignable state pairs* [124] can also be obtained symbolically using standard LIF manipulations.

5.3.3 Definite Machines and Finite-Memory Machines

The LIF representation of a sequential function brings out some other interesting aspects of finite state machines related to their *memory span*, i.e. the amount of past history needed to determine its future behavior. The following definitions are from the classic text on finite state machines by Kohavi [93]:

Definition 13: A sequential machine M is called a *definite machine of order* μ, if μ is the least integer so that the present state of M can be determined uniquely from the knowledge of the last μ inputs to M.

Definition 14: A sequential machine M is defined as a *finite-memory machine of order* μ, if μ is the least integer so that the present state of M can be determined uniquely from the knowledge of the last μ inputs and the corresponding μ outputs.

Note that while a finite-memory machine involves a limited memory span with respect to input-output sequences, a definite machine involves a limited memory span with respect to input sequences only.

Given the correspondence of an LIF representation to a reverse DFA, it is very natural that any relationship with respect to the *last* μ *inputs* shows up very clearly in the representation itself. For example, it is straightforward to observe that an acyclic LIF representation, i.e. where there are no cycles of FDs when following transitions from LIBDD terminal nodes to FDs, corresponds to a finite input span. Thus, if each state transition function of a sequential machine has an acyclic LIF representation, then the machine is definite. Furthermore, the length of the longest path in the LIF representation (over all state transition functions) provides μ, the order of definiteness of the machine.

The correspondence of acyclic LIF representations to definite machines can also be seen clearly from the canonical realization of a μ-definite machine [93], shown in Figure 5.8. It shows an

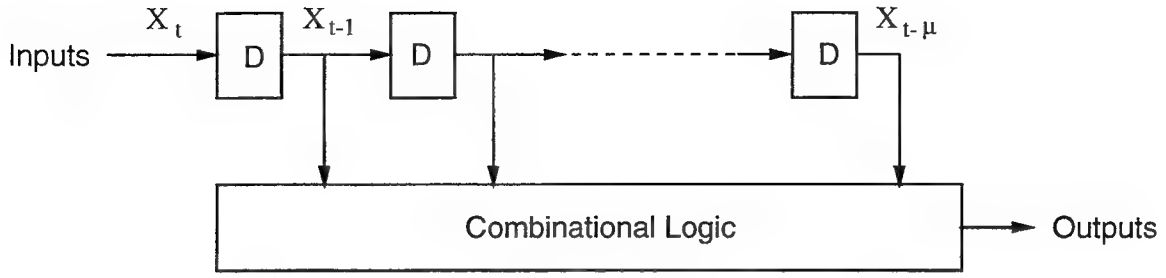


Figure 5.8: Canonical Realization of a μ-Definite Machine

array of μ delay elements (marked D), which store the last μ inputs to the machine. The circuit outputs (as well as the states) can then be implemented by a combinational circuit utilizing the stored input values. An acyclic LIF representation can be viewed in a similar way. Each terminal node of an LIBDD with a pointer to an FD can be regarded as a delay element since it implicitly denotes a parameter substitution $t - 1/t$. Furthermore, the rest of the LIBDD can be regarded as computing a combinational function in terms of the inputs, since the underlying graph is acyclic. Thus, the LIF representation includes in itself the canonical realization for a definite machine. What makes it different, and more expressive, is the presence of Basis BDDs. Recall that in the case of state transition functions, these Basis BDDs encode the initial state information, which is needed in the general case in order to determine the present state of a machine.

As for the finite-memory machines, since an LIF view of a finite state machine represents each state transition function and output function individually, the relationship from outputs to states is not emphasized at all. Thus, there is no straightforward way to determine whether a machine is finite-memory or not directly from the LIF representations.

5.4 LIF Representation vs. State Transition Graph

Though it is useful to understand the conceptual correspondence of LIF representations with DFAs, at a more detailed level, there are significant differences between the LIF representations and the traditional state transition graph representation of DFAs.

In order to support symbolic manipulation of a dynamic system of sequential functions, where new functions are continually added to the system, the emphasis in the LIF framework is on representation of individual functions. This is reflected in the fact that an FD is used per sequential function. This naturally offers the advantages of *decomposition* for a DFA, where only the relevant parts of a state space are represented on a per-output basis. Indeed, the LIF representations (FDs) for two independent outputs will be naturally decoupled even if the user provides a state space description of their product.

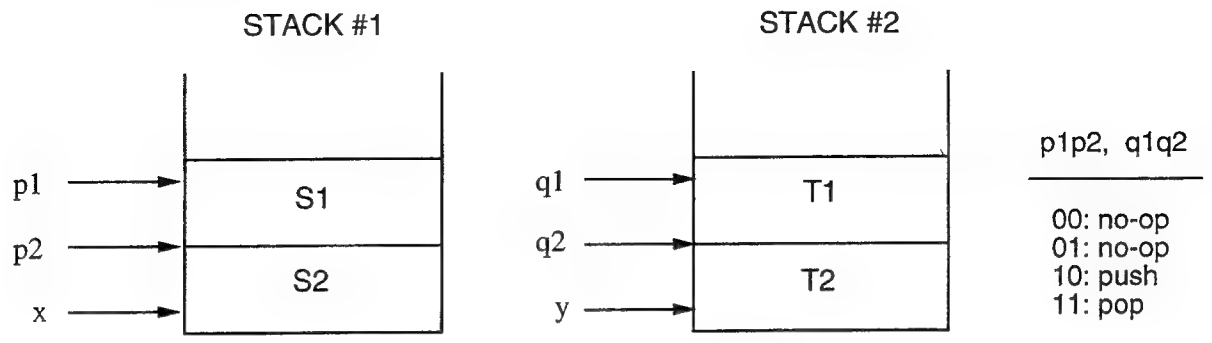


Figure 5.9: A System of Two Stacks

The decomposition property can be exploited in a traditional state transition graph representation of a DFA also. This is accomplished by minimizing the state space per individual output. However, in this case, unless the system is naturally decomposable, the information regarding states shared with other outputs is lost. In other words, in the traditional state transition graph representation, one can exploit either decomposition or state-sharing, but not both[4]. In the LIF representation, on the other hand, the minimal states (FDs) are associated with tags denoting the Boolean combinations (meta-BDDs) of the state variables (user-given LIFs) that they represent. This information is enough to allow state-sharing among different outputs, while retaining an individual representation per output.

Example 20: As an illustration, consider a system with two 2-bit deep stacks as shown in Figure 5.9. (The depth of 2-bits is for the purpose of exposition only, the following argument holds for any n-bit deep stack. Note also that this example deals with only the temporal behavior of a fixed-sized stack circuit, unlike the parametric-sized stack circuits described in the previous chapter.) Each stack has two state bits ($S1, S2$ and $T1, T2$ for Stack #1 and #2, respectively), one data input (x and y, respectively) and two control inputs ($p1, p2$ and $q1, q2$, respectively – 00, 01:no-op, 10:push, and 11:pop). It is assumed that both stacks are well-behaved, i.e. the stacks are never pushed when full and never popped when empty.

Now consider a minimal DFA representation of this system in the form of a traditional state transition graph. Clearly, it has 16 states denoting all possible combinations of the four state bits, as shown in Figure 5.10. (For ease of understanding, the global state space has been depicted as a product of the state spaces of the two stacks, differentiated by italicized labels for Stack #2. Also, for better readability, the output labels for all transitions leaving a state are shown inside the state itself.) Since the two stacks are independent, the global state space can be easily decomposed by minimizing per stack, resulting in representations shown in Figure 5.11 Part(a) for Stack #1 (Stack #2 is similar). This provides a reduction down to 4

[4]without doing significantly extra work

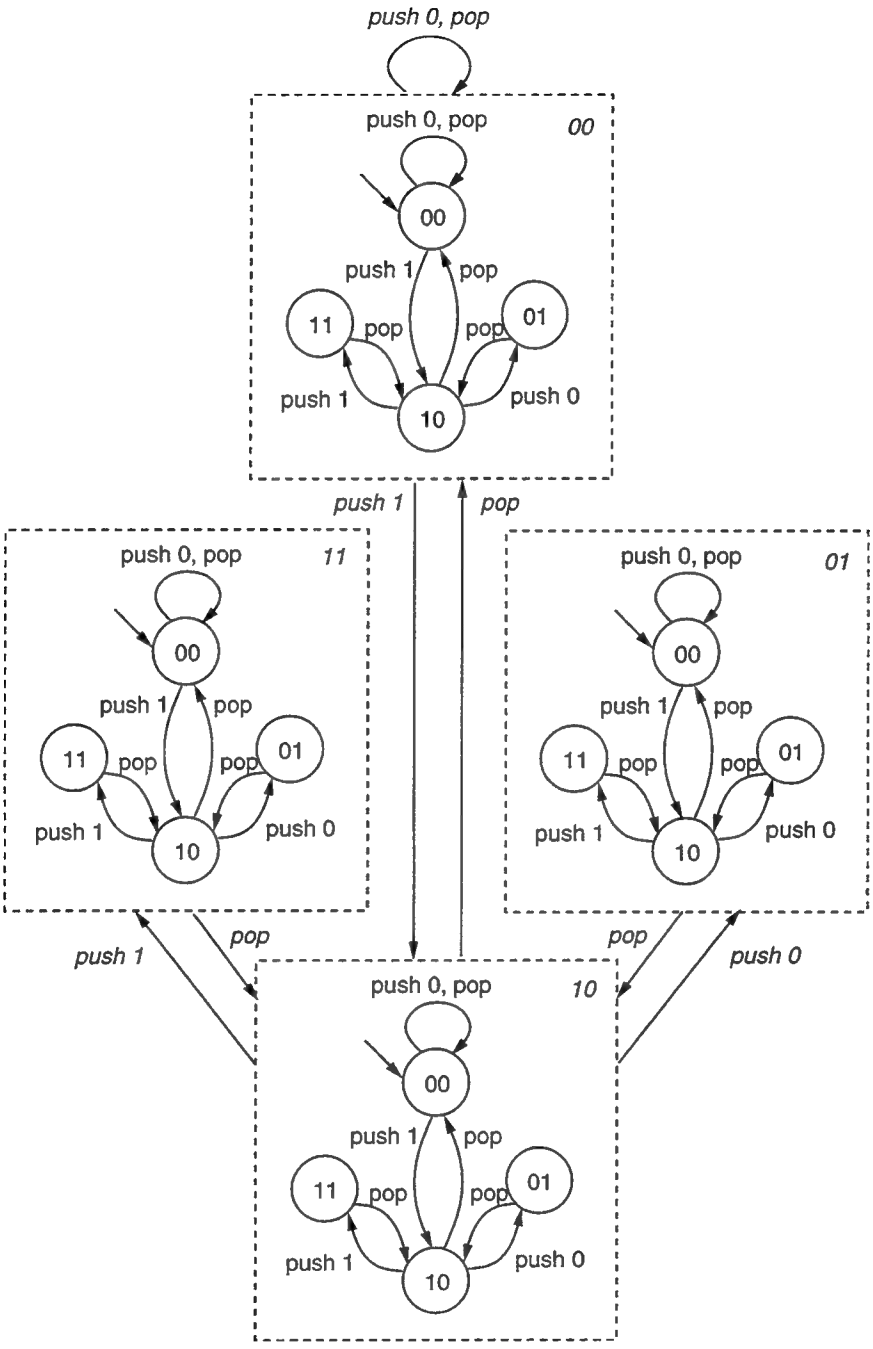
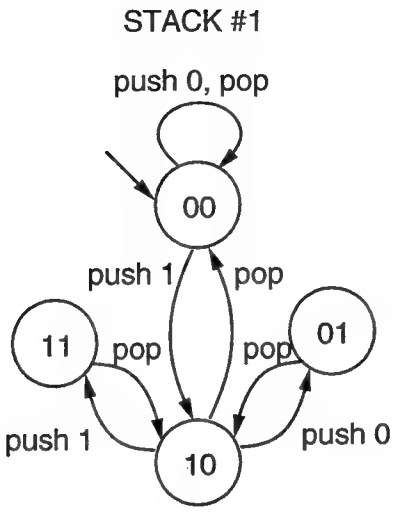
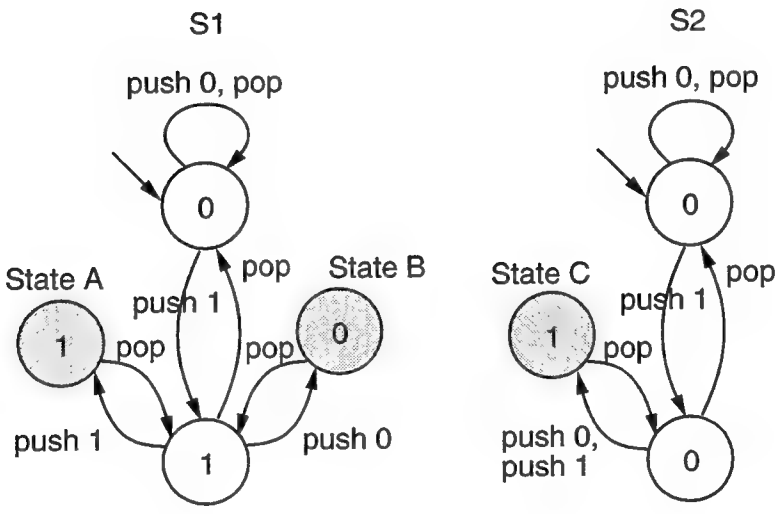


Figure 5.10: Global State Transition Graph for the Stack System



Part (a)



Part (b)

Figure 5.11: Minimized State Transition Graphs for the Stack System

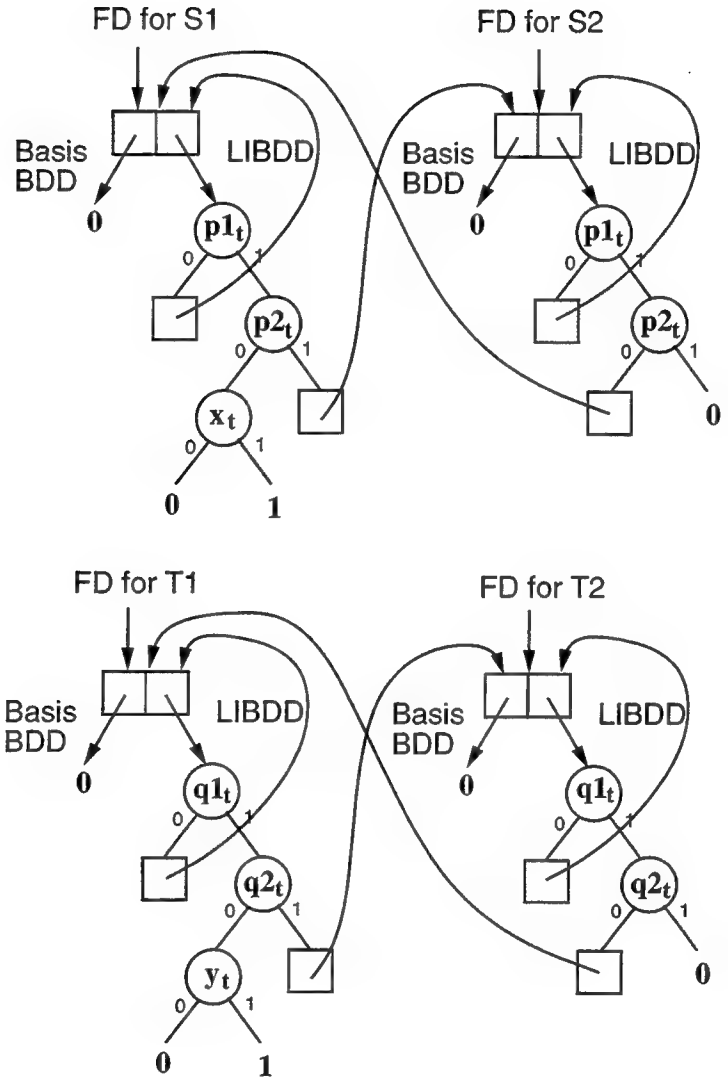


Figure 5.12: LIF Representations for the Stack System

states per stack, i.e. 8 states in all. When the same exercise is repeated for each state bit within a stack, the representations shown in Figure 5.11 Part(b) are obtained. (Bits $t1$ and $t2$ of Stack #2 are similar.) Note that since the system is not naturally decomposable along each state bit, the minimization process loses the state-sharing information, resulting in an increase in the total number of states required, now 14. Such state-sharing information is essential not only for compact representations, but also for subsequent symbolic manipulation of these functions. For example, State C of $s2$ is equivalent to states $A \vee B$ of $s1$, i.e. on any input string State C is reached in $s2$ whenever either State A or State B is reached in $s1$. Ideally, one would like to retain the advantages of decomposition, without giving up the advantages of state-sharing. (That such state-sharing exists in practice is shown by the results reported in the next section.)

On the other hand, consider an LIF representation of the entire system as shown in Figure 5.12. Note that FDs for the two stacks are naturally decoupled, since they are independent. Furthermore, within each stack, state-sharing exists between $s1$ and $s2$ (also, between $t1$ and $t2$) resulting in a final representation with only 4 FDs. It is important to note that this decomposition and state-sharing, captured by the LIF representations, are properties of the underlying sequential behavior with respect to the inputs, and not of any particular state encoding of the system description. Thus, the LIF representation naturally allows both decomposition and state-sharing, potentially resulting in substantial memory savings.

Another potential for memory savings is due to the symbolic machinery now available with standard BDD packages [18]. For example, a negation bit flag can be used to denote complementation. Just as in BDDs, a complemented LIF is symbolically represented by using a negative pointer into the representation of the uncomplemented LIF. For example, in Figure 5.3, although the minimal reverse DFA for output $s2$ has 4 states, there are only 2 FDs in the representation due to the sharing allowed by the negative edge (pointer with a dark circle).

5.5 Forward DFA vs. Reverse DFA

Returning back to the conceptual level, given that the LIF representation corresponds to a minimal reverse DFA, the next natural question is: How does this compare with a minimal *forward* DFA, i.e. the minimal classic DFA?

In theory, a reverse DFA may be exponentially more compact (or, bigger) than a forward DFA. The reason can be easily seen as follows. As mentioned earlier, in principle, a reverse DFA can be obtained from a forward DFA by reversing all transitions, and exchanging initial states with final states. This process may lead to nondeterminism, e.g. two edges into a state on the same input will now become outgoing edges on that input. Further determinization may therefore involve an exponential blow-up. However, an exponential blow-up in one direction implies an exponential compaction in the other. For example, for the language $\mathcal{L} = (0+1)^*1(0+1)^k$ (all $0/1$ strings where the last $(k+1)^{th}$ input is a 1), a minimal $DFA(\mathcal{L})$ requires at least 2^k states.

This is because it needs to check all possible k-length strings after a 1. Now consider a DFA for the reverse language $\mathcal{L}^R = (0+1)^k 1(0+1)^*$ (all 0/1 strings where the first $(k+1)^{th}$ input is a 1). Clearly a minimal $DFA(\mathcal{L}^R)$ requires only $k+3$ states in order to check the $(k+1)^{th}$ input and either accept or reject. Thus, the reverse DFA for this example is exponentially more compact than the forward DFA.

Apart from the theoretical insight, it is instructive to examine this issue in practice also. Results for obtaining canonical LIF representations for some example circuits are shown in Table 5.2, with measurements on an SGI Indigo with 32 Mb. main memory. In addition to the CPU time needed, the number of latches, inputs, outputs, unique LIFs, and sharing quotient is also reported. The number of unique LIFs is the total number of LIFs (minimal states) needed for a representation of all outputs with state-sharing. The sharing quotient indicates the quotient of the number of LIFs with, and without, state-sharing between outputs. A low number indicates a large amount of state-sharing.

The examples in the top part of the table are from the MCNC sequential benchmarks [101] consisting mostly of controllers. The examples in the bottom part are 1-bit slices of simple gate-level datapath circuits of different sizes, much like the examples described in the thesis. As can be seen from the column with the number of unique LIFs, there is a blow-up in most of the MCNC circuits[5], i.e. unique LIFs $> 2^{Latches}$. On the other hand, for the datapath circuits, the number of unique LIFs is linear in the circuit size. Note that the minimal classic DFA for each of these datapath circuits is (approximately) of size $2^{Latches}$. Thus, for these cases, *the LIF representation actually provides exponential compaction in comparison to the classic DFA representations.* Such an exponential compaction is very useful since these datapath circuits are known to be especially problematic with regard to state explosion.

Apart from compaction in the final representations, note that standard techniques for obtaining a reverse DFA rely on an explicit representation of the forward DFA. Since the forward DFAs are exponentially bigger for these datapath circuits, such techniques are prohibitive in practice due to the exponentially bigger intermediate representations. On the other hand, recall that the LIF canonicity algorithm for obtaining these compact reverse DFAs does not require the explicit representation of the forward DFAs. Though its worst-case complexity is also exponential, it performs much better in practice, as demonstrated by the results for datapath circuits. In this sense, *the LIF canonicity algorithm can be regarded as an alternative attack on the complexity of state-space representations.* It results in substantial memory savings for the circuit sizes usually encountered in practice. The benefits of state-sharing between outputs can also be clearly seen in most examples, as evidenced by the sharing quotient column.

One of the possible explanations for the observed difference between these controller and datapath circuits can be understood by drawing a parallel between LIF representations and

[5]Though the number of unique LIFs shown here is the total for all outputs, a blow-up was observed in the numbers for individual outputs too, thereby indicating that lack of decomposition is not its primary cause.

Circuit	Latches	Inputs	Outputs	CPU Time (sec.)	Unique LIFs	Sharing Quotient
dk27	3	1	2	1.1	13	0.76
dk17	3	2	3	1.2	27	0.43
lion9	3	2	1	0.9	6	1.00
beecount	3	3	4	1.4	15	0.50
dk14	3	3	5	1.5	17	0.35
dk512	4	1	3	1.3	36	0.50
ex3	4	2	2	1.3	39	0.52
ex6	4	5	8	2.0	68	0.15
shift reg 16	16	18	16	0.3	16	0.06
shift reg 64	64	66	64	0.5	64	0.02
shift reg 256	256	258	256	4.7	256	0.01
shift reg 1024	1024	1026	1024	123.4	1024	0.01
reg file 16	16	8	16	0.7	16	1.00
reg file 64	64	10	64	1.9	64	1.00
reg file 256	256	12	256	24.7	256	1.00
reg file 1024	1024	14	1024	458.1	1024	1.00
stack 8	11	3	10	0.4	16	0.20
stack 32	36	3	34	1.2	64	0.06
stack 128	135	3	130	15.3	256	0.02
stack 512	521	3	514	247.5	1024	0.01
FIFO 8	11	3	10	0.8	15	0.23
FIFO 32	36	3	34	1.7	63	0.06
FIFO 128	135	3	130	19.1	255	0.02
FIFO 512	521	3	514	282.9	1023	0.01

Table 5.2: LIF Representation of Sequential Circuits

Unique LIFs: total # LIFs for representation of all outputs with state-sharing
Sharing Quotient: # LIFs with state-sharing ÷ # LIFs without state-sharing

BDDs. LIF representations can be viewed as layered BDD-like graphs, with the layer of t-instance inputs placed before the layer of $(t-1)$-instance inputs. It is well known that smaller BDDs are obtained by placing the more "important" variables (with more fan-out, controlling variables etc.) first in the variable ordering [22]. Translating this requirement for LIFs implies that smaller LIF representations are obtained if t-instance inputs are more critical than $(t-1)$-instance inputs, i.e. if outputs depend more critically on more recent inputs than on less recent inputs. It is quite likely that this temporal preference is not exhibited by the controller circuits experimented with, thereby resulting in bigger LIF representations. On the other hand, for the datapath circuits under consideration, the recent inputs seem more important, e.g. for a FIFO of length k, only the last k data inputs are relevant. Thus, the LIF ordering is well-suited for these circuits, resulting in compact representations.

Note that though the above argument of temporal preference bears some resemblance to the earlier description of definite machines (and finite-memory machines), they are not exactly alike. In principle, it is possible for definite machines to have large LIF representations, even though they have a limited memory span with respect to past inputs. A contrived example could consist of an acyclic LIF representation where all 2^{2^u} possible combinations of u state variables are accessible from the starting FD. Clearly, such a representation is exponentially bigger than the forward DFA, which can be at most 2^u in size. Similarly for the other argument, it is also possible for indefinite machines to have compact representations. Thus, size of the LIF representation is not necessarily indicative of the memory span of the associated sequential machine.

Chapter 6

Formal Verification Applications

Since the inductive characterization of IBFs can be used to capture both structural induction in size-parametric hardware, as well as temporal induction in sequential systems, the formal verification applications of symbolic IBF manipulation naturally fall under these two broad categories.

In the domain of size-parametric hardware, apart from canonical representation, IBF manipulation can also be used for checking functional correctness for all circuit sizes by induction. Typically, the correctness property is expressed as an IBF formula. Checking it by induction corresponds to tautology-checking, which can be done automatically by obtaining its canonical representation. This method is particularly useful for an algebraic approach, where a correct circuit is characterized in terms of size-independent axioms.

For finite state sequential systems, the LIF representation provides a canonical form for an output, which is independent of the state encoding and the number of states in the system. This can be used to check input/output equivalence of finite state systems. Given the correspondence of LIF representations to reverse DFAs described in the previous chapter, it is easy to see that the LIF traversal-based method for checking equivalence corresponds to a symbolic backward traversal on the underlying state transition graphs. Though this is similar to other symbolic backward traversal techniques [55, 94], the previous work on this subject did not address the importance of reverse DFAs, or their relative advantages with respect to classic DFAs. Similar to the case of LIF representations, the LIF equivalence check is especially useful for typical datapath circuits where the exponential compaction of reverse DFAs helps to alleviate the state explosion problem.

The unified framework of LIFs for dealing with space and time opens up many interesting possibilities of combining induction in the two domains also. Its application to a hierarchy of successively more difficult verification tasks can be illustrated in the context of verification of pipelined circuits. At the simplest level, a verification task may involve checking

the equivalence of a combinational, space-parametric version of a circuit against a pipelined, time-parametric version. Given the unified framework for representing space and time, checking such an equivalence is trivial. At the next level, the space and time parameters could be combined within a multiple parameter IBF framework to check the behavioral equivalence of an un-pipelined circuit against that of a pipelined circuit for all circuit sizes. Recall that a multiple parameter IBF representation is like a minimal "multi-dimensional" automaton, and can therefore be used to check for equivalence. Finally, it is possible to use composition on space-parametric LIFs to perform a fixed-cycle symbolic simulation along the time dimension. If the sequences of LIFs so produced converges, it translates to a language containment relationship, which is also the core of other automata-based efforts for inductive verification [6, 96, 130, 131, 155]. On the other hand, if the sequence of LIFs does not converge, it may still be possible to check correctness for at least a fixed-depth of pipelining.

In this chapter, these applications are described, along with practical results from the prototype LIF package (described in Chapter 3). The experiments for all results reported in this chapter were conducted on an SGI Indigo workstation, with 32 Mb. main memory. Some practical issues are also described – depth-first vs. breadth-first strategy for LIF comparison, lazy vs. eager approach for LIF generation, and effect of variable orderings.

6.1 Functional Verification of Size-Parametric Hardware

6.1.1 Canonical LIF Representations

For size-parametric hardware, the circuit outputs are captured as IBFs defined over circuit size parameters (as described in detail in Chapter 4). Recall that the IBF representations, for both the classes of LIFs and EIFs, can be regarded as inductive extensions of BDDs, where canonicity can be ensured by using an equality check with a built-in proof by induction. Thus, the IBF representation can be directly used to check for equivalence of two parametric circuits defined in terms of the same circuit size parameters.

In Table 6.1, practical results are shown for LIF representation of some common parametric combinational circuits (already described in this thesis). The table shows both the memory requirements and the CPU time required to obtain canonical LIFs representing the circuit outputs. Since the degree of mutual-recursion in such LIFs is typically small, the effort for enforcing canonicity is minimal, and fairly good performance is obtained. The same table also shows the memory requirements for a BDD representation of the corresponding 4-bit circuit slice. Note that the LIF requirements are quite modest in comparison. Note also that LIFs are especially effective in capturing *all* outputs of multiple-output parametric circuits, such as the register cell outputs of the parametric register file.

Circuit	IBF Representation		BDD (4-bit Slice) Memory (Kb)	
	CPU Time (sec.)	Memory (Kb.)	MSB	All Bits
i-bit Ripple Carry Adder	0.6	1.1	0.7	1.1
i-bit Serial Parity Circuit	0.7	0.3	0.2	0.2
i-bit Comparator	0.6	0.5	0.5	0.5
i-bit Multiplexor	0.7	0.4	0.7	0.7
i-bit Decoder	0.9	0.7	0.2	0.8
Register File: i-bit Address, j-bit Data				
Register Cell Outputs	0.9	2.5	0.7	44.3
Read-port Outputs	0.8	3.5	1.9	7.7

Table 6.1: LIF Representation of Parametric Combinational Circuits

6.1.2 Functional Property Verification

Suppose a functional correctness property for a size-parametric circuit can be expressed as an IBF formula defined over circuit outputs represented as IBFs. Performing a proof of correctness by induction on the circuit size parameter corresponds to *tautology-checking* of the IBF formula, i.e. checking whether the IBF formula is true or not, for all combinations of the inputs and for all values of the circuit size parameter. This correspondence can be seen for the class of LIFs as follows.

Let the correctness property be expressed as g, where g is a Boolean combination on a set of LIFs $f_1, f_2, \ldots f_n$, that represent the circuit outputs, i.e. $g = \mathcal{B}(f_1, f_2, \ldots, f_n)$. Note that the correctness property is to be proved for all i-instances, $i \geq 1$, of the circuit. Since g is also an LIF, proving the property for all $i \geq 1$ translates to tautology-checking of g. Recall that this can be done by checking equality of g against '1' (true) using $EQ(g,$ '1') (Figure 3.9, Section 3.2.2.2). A run of $EQ(g,$ '1') naturally translates to a proof by induction as follows:

- Basis step of the induction: Check that the 1-instance circuit satisfies g.
  This translates to checking that $g^1 = \mathcal{B}(f_1^1, f_2^1, \ldots, f_n^1)$ is equal to '1', and is checked in Step 4 of the EQ algorithm.

- Induction hypothesis: Assume that the $(i-1)$-instance circuit satisfies g (all j-instance circuits satisfy g, $1 \leq j < i$).
  This translates to the assumption that $g^{(i-1)} = \mathcal{B}(f_1^{(i-1)}, f_2^{(i-1)}, \ldots, f_n^{(i-1)})$ is equal to '1' (all g^j are equal to '1' for $1 \leq j < i$).

- Induction step: Prove that the i-instance circuit also satisfies g.
  This translates to proving that $g^i = \mathcal{B}(f_1^i, f_2^i, \ldots, f_n^i)$ is equal to '1'.

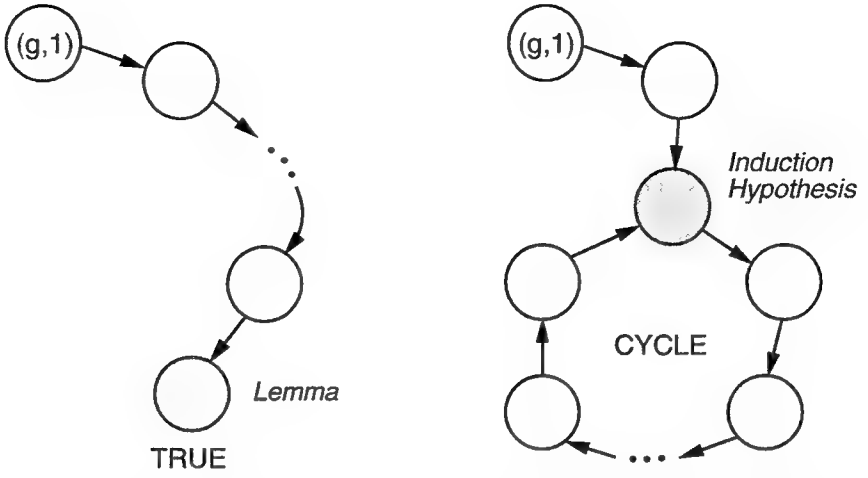


Figure 6.1: Correspondence of Comparison-graph and a Proof by Induction

Note that in the LIF representation, g^i is implicitly represented as the LIBDD for g. Step 5 of the EQ algorithm reasons about this LIBDD, where Step 5(b) may recursively lead to equality checks for other pairs of FDs. Recall that this recursion, and the associated equality dependencies between pairs of FDs, are captured by a comparison-graph shown schematically in Figure 6.1. Recall also that the top-level node (g, '1') denotes a true equivalence if it cannot reach any 'False' node in the graph. Note that for any cycle in this graph, going around the cycle translates to encountering an FD pair which implicitly represents a tautology-check for some g^j, where $j < i$, which is true by the induction hypothesis. Therefore, establishing the top-level node to be 'True' due to presence of cycles (that cannot reach nodes labeled 'False') translates to using the induction hypothesis on g^j in order to prove the induction step for g^i. In addition, any previously labeled 'True' nodes serve as useful lemmas that are needed to push through the induction. In fact, it is this very inductive reasoning that was used to prove Theorem 2, Chapter 3.

The important point is that the induction hypotheses and useful lemmas are taken into account *automatically* by the IBF method for checking equality of FDs. This not only allows complete automation of a proof by induction, but also eliminates the heuristic search typically associated with more powerful proof-theoretic frameworks. Furthermore, by including the inductive reasoning implicitly at the level of the representations, the need to perform an explicit proof by induction is avoided, thereby freeing the user from the burden of conducting it.

Example 21: Consider the task of verifying the functional correctness of a decoder, where the property to be verified is mutual exclusion of its outputs, i.e. for every i-instance decoder, check that only one of its 2^i outputs is true, rest are all false. In conventional (non-LIF) notation, this

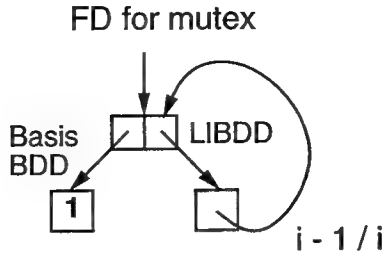


Figure 6.2: LIF Representation for the Decoder Specification

specification property can be stated as follows:

$$\forall i.\exists n.(dec^i_n \wedge (\forall m \neq n.\neg dec^i_m)),$$

where dec^i_n represents the n^{th} output of an i-instance decoder ($1 \leq n \leq 2^i$).

Recall from the LIF representation of the decoder outputs (shown in Figure 4.21, Section 4.4.2), that a binary-encoded argument J is used to represent the i-bit output index n. This trick allows a replacement of quantification over integers by quantification over Boolean-valued variables. Thus, the above specification property can be rewritten in the LIF terminology as follows:

$$\forall i.\forall A[i].\exists J[i].(dec^i(A[i], J[i]) \wedge (\forall K[i].(K[i] \oplus J[i]) \Rightarrow \neg dec^i(A[i], K[i]))),$$

where K is another i-bit binary-encoded argument used to represent the index m. This, in turn, can be translated to the following statement (by using the definitions of $\exists$ and $\forall$ in a Boolean context):

$$\begin{aligned}\forall i.((&newdec^i(A[i], J[i])|_{j^i=0} \oplus newdec^i(A[i], J[i])|_{j^i=1}))_{a^i=0} \wedge \\ &((newdec^i(A[i], J[i])|_{j^i=0} \oplus newdec^i(A[i], J[i])|_{j^i=1}))_{a^i=1}\end{aligned}$$

where $f|_{x=0/1}$ denotes the restriction of a function f for $x = 0/1$, and $Newdec$ is a new function derived from Dec by pushing in the quantifications for $J[i-1]$ and $A[i-1]$. The formula within the scope of the outermost quantification on i, can be regarded as an LIF formula, denoted $mutex^i$. Note that it is a Boolean combinations of LIFs, and its LIF representation can be automatically derived by simple symbolic manipulations on the LIF representation of Dec. The final representation is shown in Figure 6.2. Note that the specification property can now be stated as checking the truth of $\forall i.(mutex^i)$, which can be performed by conducting a tautology-check on $mutex$.

The tautology-check for $mutex$ (performed by using $EQ(mutex, `1')$) corresponds to a proof of the original specification property by induction on i as follows:

- The Basis BDD is equal to '1', signifying that $mutex^1$ is true. This provides the basis step of the induction proof.
- From the LIBDD, $mutex^i$ is true if $mutex^{i-1}$ is true. Therefore, the comparison-graph contains a single node cycle, with no 'False' nodes. Hence, the tautology-check is true. In terms of the induction proof, $mutex^{i-1}$ is true due to the induction hypothesis. Therefore, $mutex^i$ is true by induction.

Thus, the positive tautology-check for $mutex$ establishes the correctness of the decoder circuit for all sizes $i \geq 1$. In terms of a practical implementation, this check (including representation and manipulation of the decoder outputs) is performed in less than 1 second of CPU time.

6.1.3 Algebraic Specification and Verification

As described in the introduction to this thesis, an important issue with formal verification in general, is the adequacy of specifications: Do they mean what is intended? Are they are complete? Are they relevant? etc. In the particular context of the IBF methodology, some of these issues are illustrated by the example of the ripple carry adder. Given that the outputs of a parametric ripple carry adder circuit can be represented as LIFs, what can be proved about the correctness of such a circuit?

Ideally, one would like to be able to verify that a given adder circuit performs *addition*, where the addition operation is defined over natural numbers or integers. However, note that the domain of the IBF methodology is strictly Boolean. In other words, it is not possible to reason about the domain of integers (rationals, reals etc.) unless it is encoded in the form of Boolean-valued variables. So, one could specify a Boolean version of the addition operation, by using the standard binary representation of natural numbers. However, it would turn out to be trivially identical to the standard ripple carry adder circuit description. This should not really be a surprise, since that is how the ripple carry adder got designed anyway! The problem is that if both the circuit and the specification use the same encoding for binary representation, they are likely to contain the same errors. Therefore, it does not mean much to verify their equivalence.

A different approach would be to use a style of algebraic specification, where addition for natural numbers is specified in terms of *algebraic axioms* that characterize the operation. Such a style has been used extensively in software verification, especially for specification of systems, abstract data types etc. [9, 102, 153]. For the addition operation, for example, it may be specified that the operation should satisfy the law of commutativity, the law of associativity etc. These are quite straightforward to verify for the ripple carry adder circuit by using symbolic LIF manipulations. Note that the biggest advantage is that the symbolic approach allows all data values to be handled simultaneously, and the automatic inductive approach allows these

axioms to be verified for all sizes of the circuit. Clearly, this still does not address the issue of completeness of specifications. It can only be said that at minimum, any circuit which purports to be an adder should satisfy these axioms. As always, the quality of verification can only be as good as the quality of specifications.

Example 22: As a concrete example, consider verifying the algebraic property that when a binary input X is added to itself, it produces a left shift at the output. In terms of LIFs, the shifted output can be specified as follows:

for $i = 1, out^1 = 0$
for $i > 1, out^i = X_{(i-1)}$

The verification task is to show the equivalence of the LIF out to the LIF sum for the ripple carry adder (as before) where both inputs are equal to X ($A = B = X$) and the carry input is 0 ($c\_in = 0$). The LIF representation for the latter can be obtained by standard LIF manipulations for the Restriction operation, resulting in a representation as shown in Figure 6.3. Clearly, the two LIFs are equivalent, and the task takes less than 1 second of CPU time.

6.2 Behavioral Verification of Sequential Systems

Most verification applications of the IBF methodology for sequential systems exploit the fact that the LIF representation for a sequential function is canonical in terms of the strings of inputs, and it does not rely upon the number of states, or a particular state encoding. Some typical applications are:

- FSM input/output equivalence — By representing the state transition functions and the output functions of FSMs as LIFs, input/output equivalence can be ensured by checking equality of the output LIFs.
- Language equivalence/containment — For cases where the language acceptance criterion can be expressed as a predicate on the symbolic state vector (represented as an LIF), these problems reduce to LIF equivalence/implication problems respectively.
- Reachability analysis — Reachability of a particular state (or a set of states), can be checked by checking satisfiability of the LIF denoting the corresponding state predicate.
- Symbolic representation of the set of all reachable states — Since the symbolic state vector S can be represented as a vector of LIFs, the characteristic function of its range ($\Phi(S)$) [48] provides a symbolic representation of the set of all reachable states.
- Synchronous composition of FSMs/finite state automata — This can be easily obtained through standard manipulations for LIF composition.

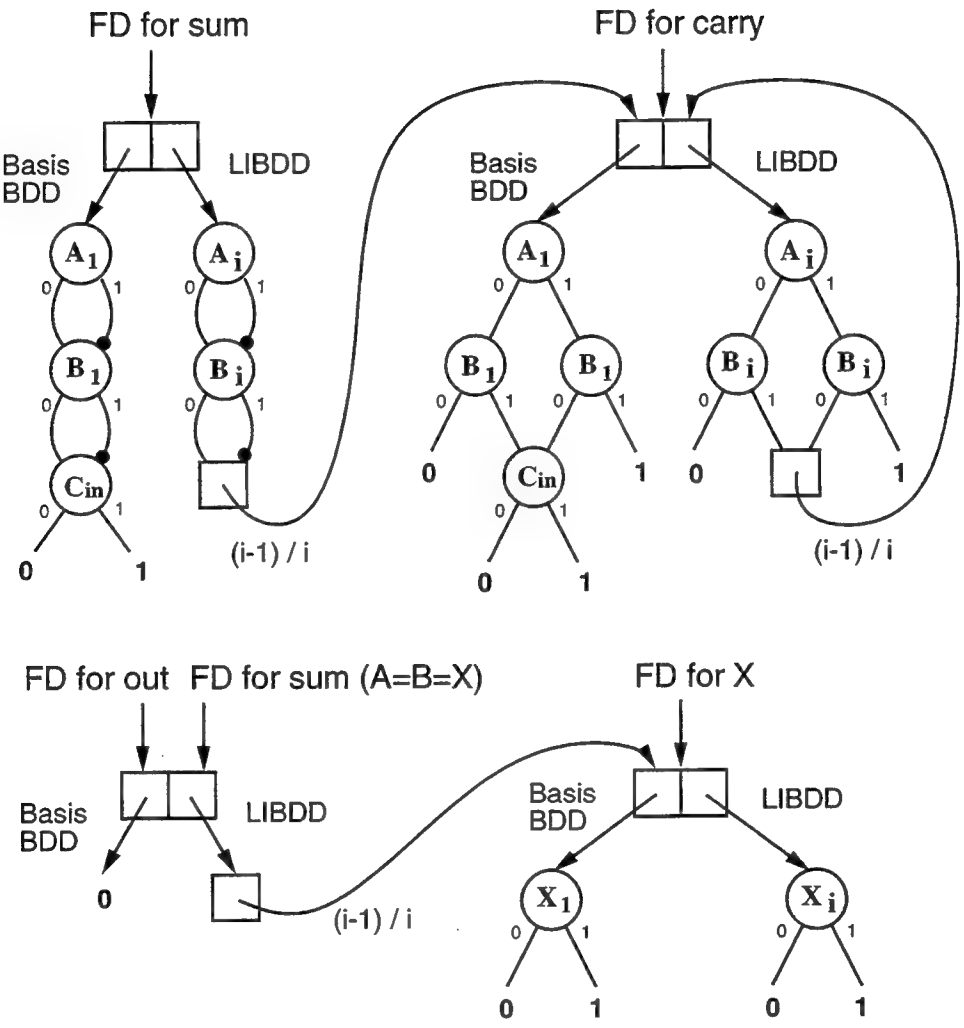


Figure 6.3: LIF Representations for Example 22

Apart from verification, symbolic LIF manipulation is potentially useful in other areas as well, such as in sequential logic synthesis, where LIFs permit easy computation and storage of input, output, and implicit don't-care sequences for direct use in synthesis.

Example 23: As a simple example of checking input/output equivalence of two FSMs using LIF representations, consider the state transition diagrams of two 2-bit counters as shown in Figure 6.4. FSM #1 (with states $s1, s2$, outputs $t1, t2$) is the standard 2-bit counter shown earlier, while FSM #2 (with states $y1, y2, y3$, outputs $z1, z2$) has a dummy state with 0-input transitions to and from the initial state. Note that the number of states and state encodings are different for the two machines.

Assume that the initial LIF representations have already been obtained from the state transition equations, and that the required combination LIFs have also been generated, i.e. the Generation Phase of the LIF canonicity algorithm is complete. In the next phase, the Comparison Phase, the FDs for all LIFs are compared pairwise in order to check for distinctness. On comparing the outputs $t1$ with $z1$, and $t2$ with $z2$, the comparison graphs obtained are as shown in Figure 6.5. (Recall that in these graphs, each node represents the equality of the marked FD pair, and each edge represents the dependence of equality of the source pair on that of the sink pair.) For example, the equality of outputs $t1$ and $z1$ requires the equality of the set of shaded states $(s1, y1)$, as well as equality of the set of unshaded states $(\neg s1, \neg y1)$. Since there are no 'False' nodes in the comparison graphs, both sets of outputs are equal, thereby verifying the input/output equivalence of the two FSMs.

6.2.1 Canonical LIF Representations: Controller Circuits

The results for obtaining canonical LIF representations of sequential circuits are not as uniform as those for the parametric combinational circuits described in the last section. This is not surprising since temporal induction in such circuits is rarely as regular as structural induction in combinational circuits. These results for the MCNC sequential benchmark circuits [101] are shown in Table 6.2. (Part of this table has already been shown in Table 5.2 in Chapter 5.) In addition to the CPU time needed to obtain canonical LIF representations for the circuit outputs, the table also shows the circuit features (number of latches, inputs, outputs, and gates), and the results for number of unique LIFs. The number of unique LIFs gives a measure of the inherent complexity of the problem, since it represents the number of distinct states in the reverse DFA representation for each output. It is easily observed that the CPU time correlates well with this measure.

As can be seen from the table, many (though not all) MCNC sequential benchmarks have been handled by the prototype implementation. Also note that the number of unique LIFs is typically greater than the possible number of states in a minimal classic DFA ($2^{Latches}$). As described in

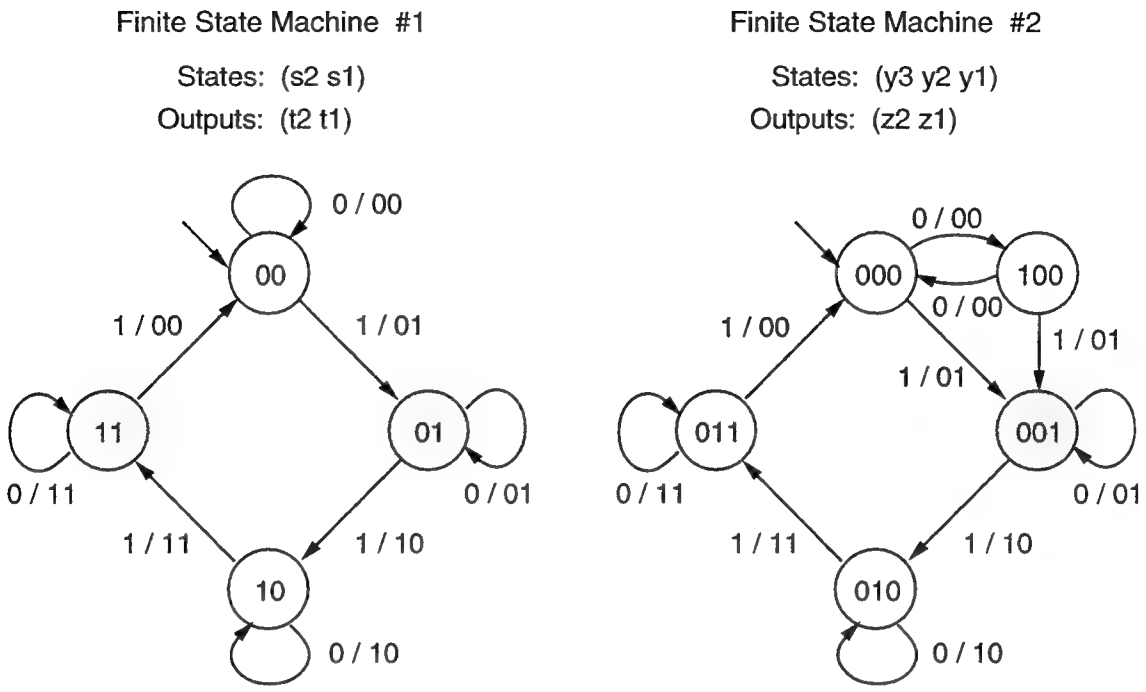


Figure 6.4: State Transition Diagrams for Two 2-bit Counters

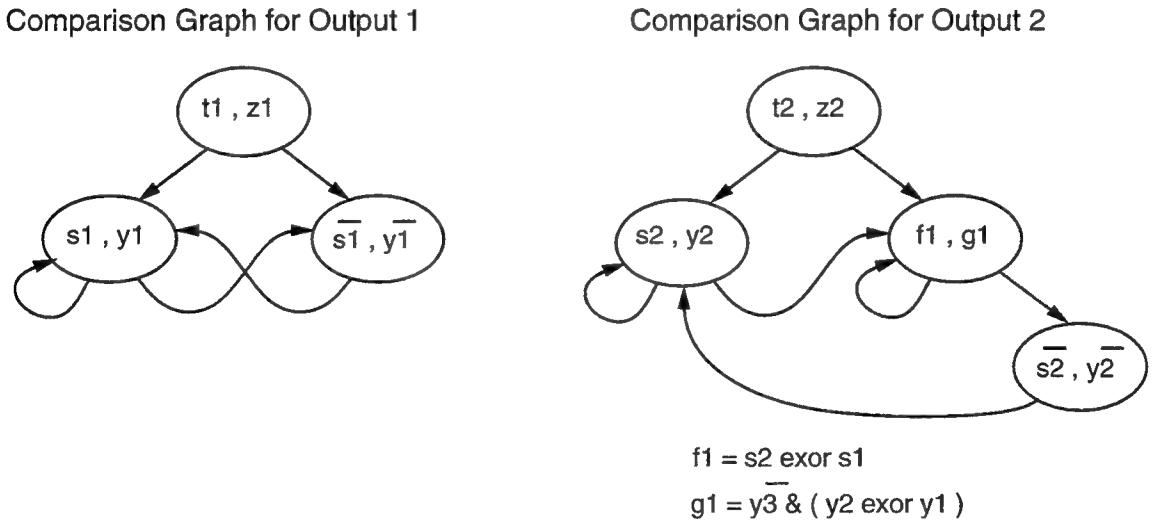


Figure 6.5: Comparison Graphs for Checking FSM Input/Output Equivalence

Circuit	Latches	Inputs	Outputs	Gates	Time (sec.)	Unique LIFs
lion	2	2	1	22	0.9	3
train4	2	2	1	21	0.9	3
dk15	2	3	5	65	1.4	10
mc	2	3	5	40	1.4	11
tav	2	4	4	34	1.2	8
shiftreg	3	1	1	22	0.9	4
dk27	3	1	2	35	1.2	13
lion9	3	2	1	37	0.9	6
ex5	3	2	2	60	1.2	30
dk17	3	2	3	57	1.2	27
beecount	3	3	4	54	1.3	15
dk14	3	3	5	86	1.4	17
modulo12	4	1	1	44	0.9	0
bbtas	4	2	2	44	1.0	14
train11	4	2	1	50	0.9	14
ex3	4	2	2	68	1.2	39
bbara	4	4	2	70	1.2	12
ex6	3	5	8	90	1.9	68
dk512	4	1	3	67	1.4	36
cse	4	7	7	163	20.6	811

Table 6.2: LIF Representation of MCNC Sequential Circuits

Chapter 5, this is possibly due to the fact that most of these circuits represent controllers that do not exhibit a temporal preference for the more recent inputs vs. the less recent ones.

6.2.2 Canonical LIF Representations: Datapath Circuits

When the same exercise of obtaining canonical LIF representations is repeated for typical datapath circuits – such as a register file, a shift register, a stack, and a FIFO – the results obtained are very different. The CPU time needed for these circuits is shown in Figure 6.6, and the number of unique LIFs is shown in Figure 6.7. Note from the graphs that the number of unique LIFs is linear in the number of latches in the circuit (circuit size). This is in direct contrast to the classic DFA representation of such circuits, which is known to be exponential.

As described previously in Chapter 5, the reason for the good LIF results for these datapath circuits is that the reverse DFA representation is exponentially more compact than the forward DFA representation for such circuits. It is not difficult to intuitively see why this should be so. For example, for a register file cell, a reverse DFA representation has to only remember the last input which it was loaded with, whereas a forward (classic) DFA has to remember the entire possible history starting from the initial state. As mentioned before, such exponential compaction is especially useful because datapath circuits are known to cause a state explosion problem in practice.

6.2.3 Behavioral Verification of Pipelined Circuits

As a simple example of verifying the sequential behavior of pipelined circuits, consider the following problem originally described by Hu and Dill [80]:

Example 24: The task is to verify the equivalence of a pipelined implementation of a moving filter for computing averages, against its combinational specification as shown in Figure 6.8. Note that this problem can be parameterized in three possible ways – time (t), the depth of pipelining (p, which also determines the number of data inputs 2^p), and the bit-width of the data inputs (w). For now, let the LIF representation for the circuit use only the time parameter t. (Later in this chapter, the bit-width parameter w is also considered as an induction parameter.) It is straightforward to use LIF equivalence-checking to verify the equivalence of the pipelined implementation against the combinational specification for fixed sizes of the circuit. These results are shown in Table 6.3 for different values of p and w.

For comparison, the table also shows the results obtained by Hu and Dill for the plain backward symbolic traversal (Column 'Backward') and their own technique (Column 'Hu & Dill'). (In this and subsequent tables, "NA" denotes that the particular result is not available.) Their technique utilizes implicitly conjoined invariants between different iterations of the symbolic

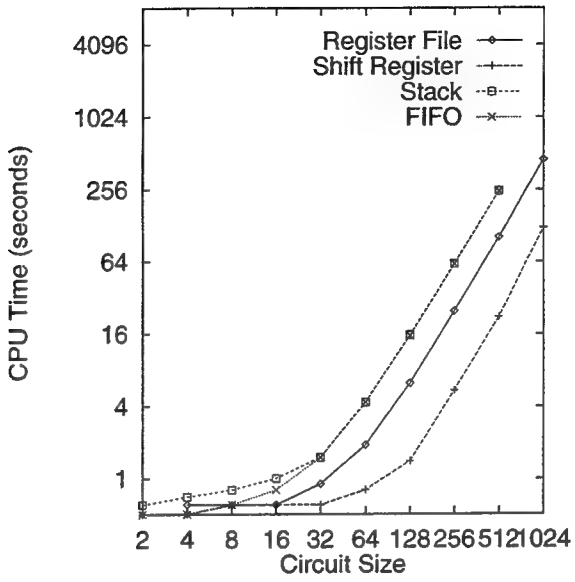


Figure 6.6: Performance Results for Datapath Circuits

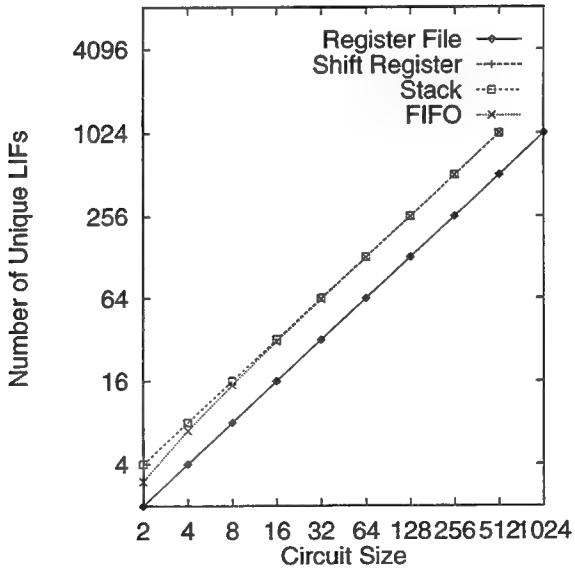


Figure 6.7: Size of LIF Representations for Datapath Circuits

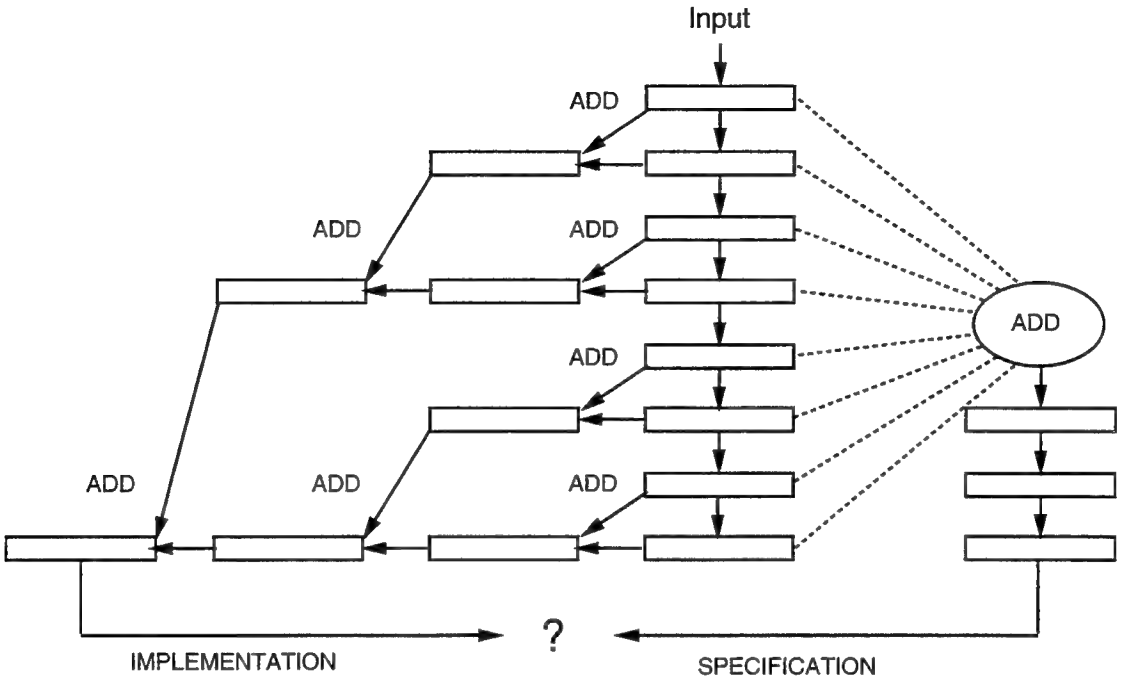


Figure 6.8: Verification of Moving Average Filter

Pipeline Stages, p	Bit-width w	Backward Time (sec.)	Hu & Dill Time (sec.)	LIF Equivalence		
				Time (sec.)	# LIF Checks	# LIF Gen
2	4	NA	NA	0.6	12	24
2	8	6	5	0.7	20	44
2	16	NA	NA	0.9	36	84
3	4	NA	NA	3.0	21	174
3	8	$>$ 30 min.	47	23.2	33	326
3	16	NA	NA	153.5	57	630
4	8	$>$ 30 min.	405	$>$ 30 min.	–	–

Table 6.3: Results for Verification of Moving Average Filter

backward traversal to prune the state-space search. From the table it appears that the LIF equivalence technique lies somewhere between the plain symbolic backward traversal technique and the technique proposed by Hu and Dill. The reasoning goes as follows. The invariants are expressed as Boolean formulas on state variables, which correspond to meta-BDDs in the LIF framework. It is quite likely that some of the effect of invariants is captured by the meta-space equivalence manipulations used by the LIF method. For example, using a meta-BDD representative for each (partially converged) LIF equivalence class may help to further establish the equivalence of two LIFs within the meta-space itself, *without having to explicitly check the equivalence of the corresponding Basis BDDs and LIBDDs.* Thus, the use of such meta-space equivalences to simplify the task of checking FD equivalence is akin to using invariants, though the exact invariants used in the two techniques may be quite different.

6.2.4 Practical Issues

In this section some practical issues are highlighted, which affect the performance of the LIF package for general symbolic LIF manipulation. These issues become crucial in the context of behavioral verification of sequential systems, which is harder than functional verification of parametric combinational circuits.

6.2.4.1 Depth-first vs. Breadth-first LIF Comparison

In the description of the EQ algorithm (Figure 3.9, Chapter 3), which is used to check the equivalence of two FDs, the check on LIBDDs (Step 5) was implemented by using a depth-first strategy for recursive checks at the LIBDD terminal nodes. For example, for two LIBDDs as shown in Figure 6.9, the depth-first order for recursive checks on pairs of FDs is: $(A, P), (B, 1), (0, Q), (C, R)$.

However, this depth-first strategy is not essential to the algorithm. (It was used mostly to simplify the exposition and the associated complexity analysis.) Instead, it is possible to use a breadth-first strategy for implementing the recursive checks. This is done by collapsing all terminal node checks for the pair of LIBDDs into one single LIF formula. Checking the truth of this formula corresponds to checking that the LIBDDs are equal. For the example of Figure 6.9, this formula turns out to be $f = (\neg(A \oplus P)) \wedge B \wedge (\neg Q) \wedge (\neg(C \oplus R))$. This is used to recursively generate a check for $EQ(f, 1)$.

Note that only Step 5 of the EQ algorithm is affected by this change. The rest of the algorithm, including the check on Basis BDDs and labeling of nodes in the comparison-graph, remains the same. Thus, Theorem 2 of Chapter 3 still applies to the comparison-graph, which is a simple chain of dependences in this case. Note also that while the difference between the depth-first and the breadth-first strategies may appear to be that between an "explicit" method

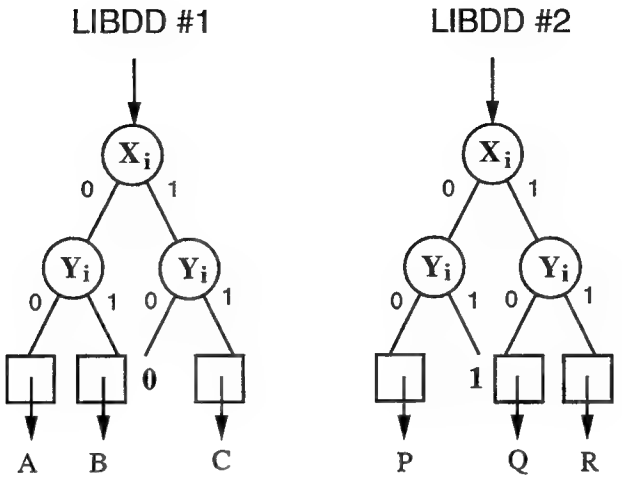


Figure 6.9: Comparison of LIBDDs

and an "implicit" method, there is a need to be more careful than that. Clearly, the breadth-first strategy is implicit. However, even the depth-first strategy is implicit in the sense that the LIBDD representation is symbolic in terms of the inputs. In fact, the depth-first recursive checks are generated by a standard BDD-like recursive traversal on the structure of the LIBDDs. Another way to view this is that the degree of "implicitness" depends on the level of abstraction being considered. At the level of inputs, both strategies are implicit. However at the level of the graph structures, depth-first is explicit while breadth-first is implicit.

These strategies were compared for different tasks, and for different kinds of circuits. For obtaining canonical LIF representations, the breadth-first strategy could not handle two of the MCNC circuits (Circuits dk14 and ex1), while the depth-first strategy did, as shown in Table 6.2. (The default strategy is depth-first, i.e. all results reported in this chapter use the depth-first strategy, unless stated otherwise.) The same effect was also observed for the case of datapath circuits and counter circuits. These results are shown in Figures 6.10, 6.11, 6.12, 6.13, and 6.14, where the CPU time is plotted as a function of increasing circuit size for different circuits. Note that the depth-first strategy is clearly better, in that it is less expensive, and it can handle bigger circuits.

On the other hand, the results for checking LIF equivalence favor the breadth-first strategy. These results are shown in Table 6.4 for those MCNC circuits where the depth-first strategy failed to either obtain canonical LIF representations, or to perform the LIF equivalence check. For the purpose of checking LIF equivalence, two different versions were created for each circuit, with different state-encodings and/or different number of states, through use of the Berkeley SIS system [133]. In the table, $\#EqualPairs$ indicates the number of LIF pairs found equal, and $\#Checks$ indicates the number of LIF equivalence checks performed.

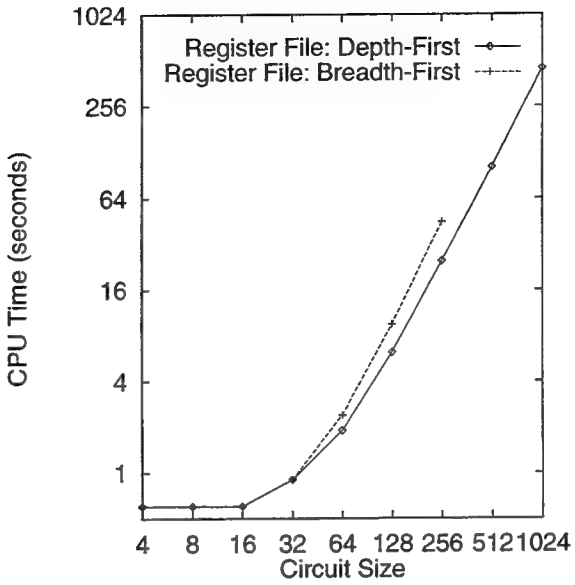


Figure 6.10: LIF Representation for Register File: Depth-First vs. Breadth-First

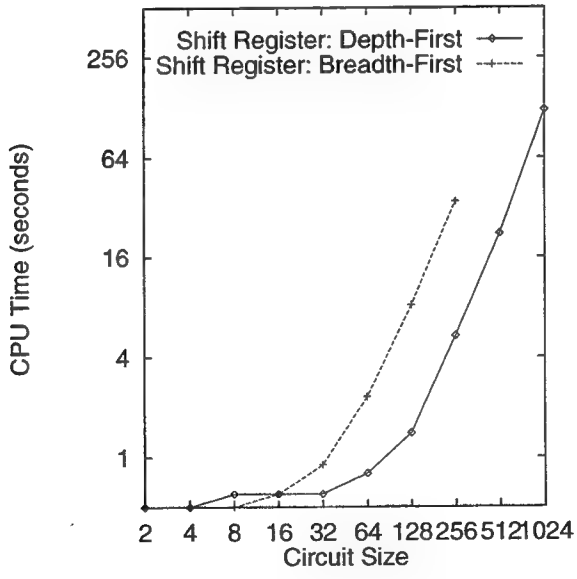


Figure 6.11: LIF Representation for Shift Register: Depth-First vs. Breadth-First

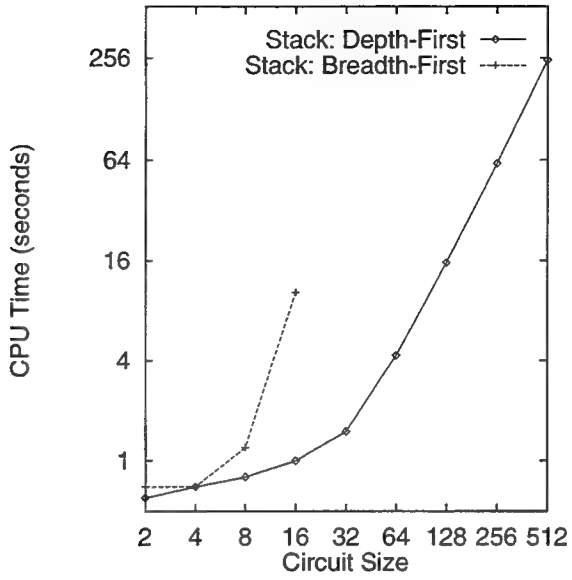


Figure 6.12: LIF Representation for Stack: Depth-First vs. Breadth-First

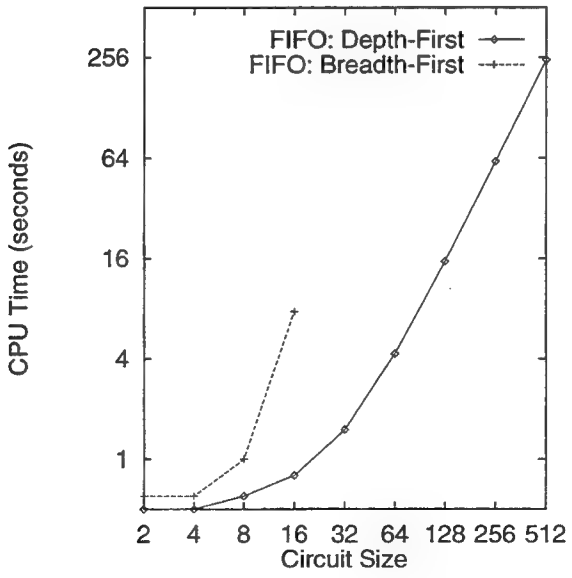


Figure 6.13: LIF Representation for FIFO: Depth-First vs. Breadth-First

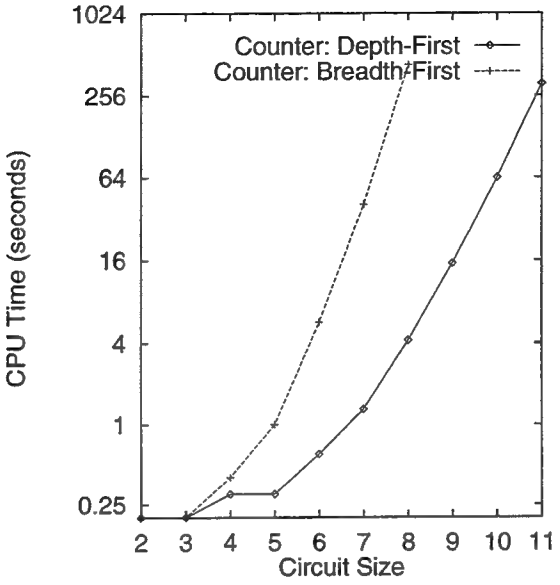


Figure 6.14: LIF Representation for Counter: Depth-First vs. Breadth-First

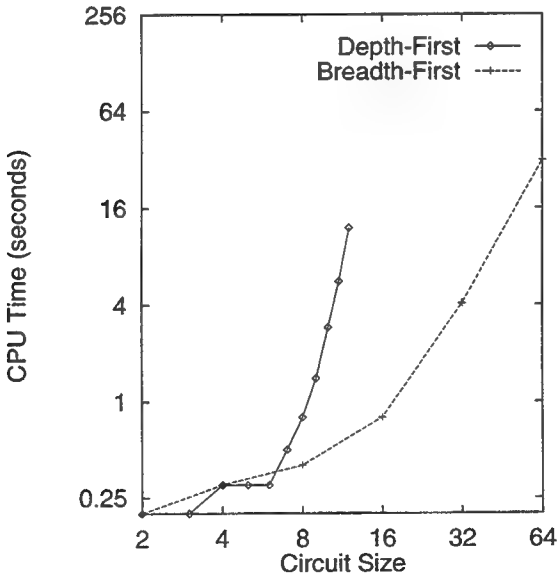


Figure 6.15: LIF Equivalence for Counters: Depth-first vs. Breadth-first

Circuits	Latches	Depth-first			Breadth-first		
		Time (sec.)	#Equal Pairs	#Checks	Time (sec.)	#Equal Pairs	#Checks
dk27	3,3	0.6	4	19	0.6	9	22
s27	3,3	0.6	1	60	0.6	4	23
dk512	4,4	–	–	–	0.8	21	56
ex3	4,4	–	–	–	0.7	10	39
ex4	4,4	–	–	–	4.5	107	616
ex6	4,4	–	–	–	1.9	37	339
ex7	4,4	0.8	3	231	0.7	8	32
s386	4,4	–	–	–	2.9	37	354
opus	4,5	–	–	–	4.1	37	707
bbsse	4,5	–	–	–	2.6	36	645
cse	4,5	–	–	–	2.7	42	446
dk16	5,5	–	–	–	2.0	16	64
ex2	5,5	–	–	–	0.9	10	42
kirkman	5,5	–	–	–	8.0	60	16672
s208	5,5	–	–	–	2.5	13	822
s420	5,5	–	–	–	2.3	14	828
keyb	5,5	–	–	–	4.7	14	1171
s1	5,6	–	–	–	81.6	36	4458

Table 6.4: LIF Equivalence for MCNC Circuits: Depth-first vs. Breadth-first

For the counter circuits also, the results for checking equivalence of two different versions favor the breadth-first strategy, as shown in the graph of Figure 6.15.

The reasons for the observed difference between the two strategies can be understood as follows. The advantage with the depth-first strategy is that partial results are available for the equivalence check on each LIBDD terminal node pair, which can be reused in the process of obtaining canonical forms for all LIFs in the system. This is in contrast to the breadth-first strategy, where a failure for the collapsed formula gives no information about the individual checks for the LIBDD terminal node pairs. Therefore, for obtaining canonical LIF representations, the depth-first strategy performs better. However, if canonicity is not desired, but the goal is only to check equivalence of two given circuits, the implicit nature of the breadth-first strategy is more useful.

6.2.4.2 Lazy vs. Eager LIF Generation

As briefly mentioned earlier, the LIF prototype implementation uses a lazy approach for obtaining FDs for newly generated LIFs. In other words, whenever a new LIF is generated to denote a Boolean combination of known LIFs, only a meta-BDD for this new LIF (with a place-holder FD) is created. The Basis BDD and the LIBDD for the new LIF, which are obtained by using function composition on the meta-BDD, are obtained only on a need-to basis. Therefore, for every new LIF, though the corresponding meta-BDD and FD always exist in the system, the Basis BDD and LIBDD are generated in a lazy manner.

The reason for adopting a lazy approach is that with the gradual convergence of LIF equivalence classes as time progresses, checking pairwise equivalence of FDs may sometimes be adequately performed within the meta-space itself. For example, for any equivalence class of LIFs, a representative meta-BDD is chosen for all further computations, which may result in the meta-space equivalence of two other FDs which were previously regarded non-equivalent in the meta-space. In such cases, checking the equivalence of the corresponding Basis BDDs and LIBDDs is *not* required, and it may be a waste of effort to generate these structures. Clearly, if a meta-space check is not enough to guarantee equivalence of two FDs, the Basis BDDs and LIBDDs are definitely required, and in this case they are generated from the meta-BDD.

To study the potential advantage of using this lazy approach, separate experiments were conducted which consisted of generating all tentative FDs *eagerly* in the system, before embarking upon the comparison of LIFs. The results comparing the lazy approach with the eager approach are shown in Table 6.5 for checking the equivalence of MCNC circuits (using the more successful breadth-first strategy described in the previous section). In the table, Gen indicates the number of LIFs generated, and $Ratio$ is the ratio of LIFs generated in the lazy approach to those in the eager approach (where successful).

For the counter circuits also, the lazy approach was clearly better, as demonstrated in Figure 6.16 for obtaining canonical LIF representations, and in Figure 6.17 for checking equivalence

Circuits	Latches	Lazy LIF Generation		Eager LIF Generation		Ratio
		Time (sec.)	#Gen	Time (sec.)	#Gen	
dk27	3,3	0.6	22	0.7	99	0.22
s27	3,3	0.6	23	0.7	80	0.29
dk512	4,4	0.8	56	0.9	621	0.21
ex3	4,4	0.7	39	0.9	300	0.13
ex4	4,4	4.5	616	38.4	23482	0.03
ex6	4,4	1.9	339	2.5	915	0.34
ex7	4,4	0.7	32	0.9	436	0.07
s386	4,4	2.9	354	12.0	3764	0.09
opus	4,5	4.1	707	27.9	7155	0.10
bbsse	4,5	2.6	645	8.6	2341	0.28
cse	4,5	2.7	446	8.8	4276	0.11
dk16	5,5	2.0	64	8.0	7393	0.01
ex2	5,5	0.9	42	4.1	4548	0.01
kirkman	5,5	8.0	16672	–	–	–
s208	5,5	2.5	822	57.7	98772	0.01
s420	5,5	2.3	828	3.9	2799	0.30
keyb	5,5	4.7	1171	–	–	–
s1	5,6	81.6	4458	–	–	–

Table 6.5: LIF Equivalence for MCNC Circuits: Eager vs. Lazy

between different circuits. (In these experiments, the better strategy was used for each task, i.e. depth-first for the former, and breadth-first for the latter). However, for the datapath circuits (register file, shift register, stack and FIFO), not much difference was observed between the lazy and eager approaches. This is because their representations largely involve user-given LIFs only, and very few new LIFs are needed, making LIF generation a non-issue.

6.2.4.3 Effect of Variable Orderings

Along with inheriting the significant advantages of BDDs, the IBF representations have inherited their limitations too. One such concern is the effect of variables orderings on the size of the resulting and intermediate representations. Good heuristics are needed for ordering the parameterized variables (in the Basis BDD and the LIBDD Boolean spaces), as well as the meta-variables (denoting the user-defined LIFs in the meta-space). Since the same parameterized variable ordering is used at every level of recursion in the IBF equality-checking algorithm (EQ), the problems with a bad ordering may get compounded, resulting in a potential explosion. For the meta-space also, a bad ordering may result in significantly more LIFs being generated, since the FD (Basis BDD and LIBDD) is obtained by function composition on the recursive structure of the corresponding meta-BDD. Thus, the user-provided orderings for both the parameterized variables and meta-variables can significantly affect performance, even if the final representations are compact.

The experiments reported so far in this chapter have used a standard scheme [58, 105] for ordering input variables according to their depth from the primary outputs. In order to study the effect of variable orderings in these circuits, some random orderings were also tried. However, these did not make much of a difference. In the case of MCNC circuits, the successful circuits were small enough anyway, and the rest could not be handled with random orderings either. A potential extension is to use the dynamic variable reordering scheme designed by Rudell [132] to automatically choose a satisfactory ordering where possible. For the case of datapath circuits, since the representations did not require Boolean combinations of user-given LIFs, the meta-variable ordering had no effect, and there are very few parameterized inputs. For counter circuits, some difference was observed, as shown in Figure 6.18. Given the high complexity of this task, a rough factor of 1.5 difference is relatively minor. Thus, though the issue of good orderings is probably important, more experiments are needed to ascertain any extra issues that may be involved, over and above the general BDD heuristics.

6.2.5 Related Work

Given the correspondence of LIF representations and reverse DFA representations of sequential systems, an LIF traversal can be regarded as a backward traversal of the state-space. Similar to

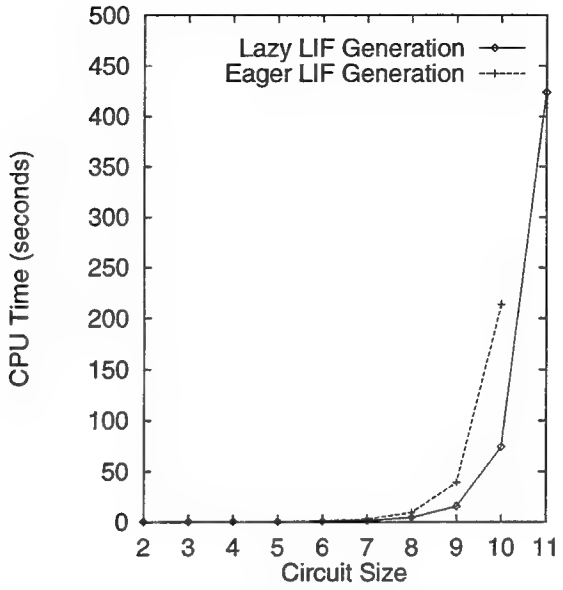


Figure 6.16: LIF Representation for Counter Circuits: Eager vs. Lazy

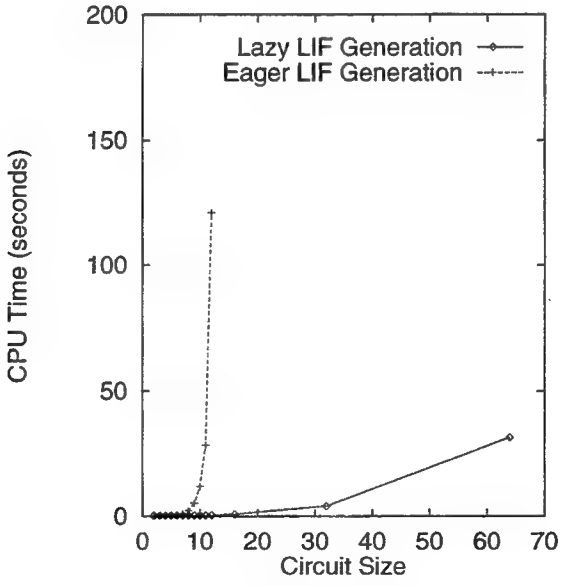


Figure 6.17: LIF Equivalence for Counter Circuits: Eager vs. Lazy

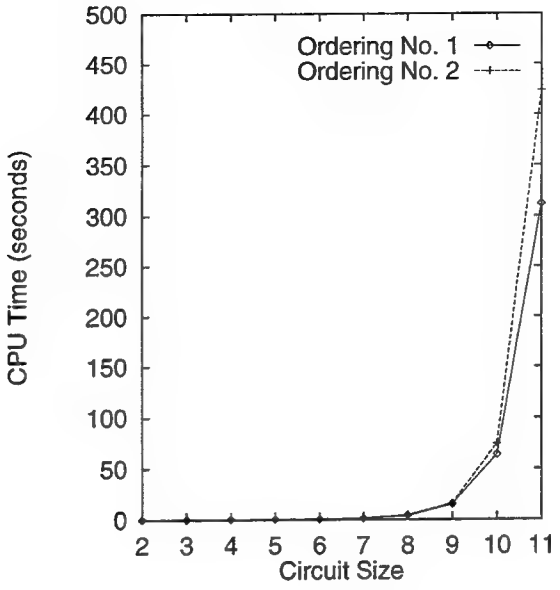


Figure 6.18: LIF Representation for Counter: Variable Orderings

the case of forward traversal, this can be either an explicit depth-first traversal, or an implicit, symbolic breadth-first traversal. However, the reversal in direction gives it quite a different flavor from the more popular symbolic forward traversal techniques [31, 48, 108]. The latter typically start from the initial state, and implicitly obtain the symbolic representation of the next-states satisfying a particular property. On the other hand, LIF-based verification techniques typically start from a sort of final state (satisfying some property), and work backwards implicitly through the transitions. For example, in verifying the I/O equivalence of the two 2-bit counters of Example 24, each comparison graph of Figure 6.5 effectively captures a symbolic backward traversal starting from states where the outputs are identical, and ensuring that they remain so in the predecessor states.

In terms of other techniques based on symbolic backward traversal [55, 94], there is a strong similarity between them and the LIF-based method of checking I/O equivalence. In fact, the implicit breadth-first version of the LIF equality checking algorithm (described in the previous section) is the operational-dual of Filkorn's method based on computing non-equivalent classes of states [55]. However, at the higher level, it is easy to view the LIF algorithm as an induction proof, in that a general purpose equality checking algorithm is used which involves checking the basis conditions and induction hypotheses. There does not arise any need to view it in terms of states and transitions. Admittedly, the same inductive reasoning can also be used to justify Filkorn's method. The difference is that it takes place at the meta-theoretic level in his method, while in the LIF framework it is incorporated into the operational method.

Furthermore, the application of the LIF algorithm is not limited to a single verification task, but

its intermediate results are useful for general-purpose symbolic manipulation of the sequential systems. In some sense, this can be viewed as a space-time tradeoff. At one end, the LIF algorithm is geared towards working directly on the entire length of the input strings. In contrast Filkorn's BDD-based method iterates sequentially on inputs, and can possibly require re-computation of intermediate results. However, it is this very feature that allows LIF-based symbolic manipulation of sequential functions in the same manner as BDD-based symbolic manipulation of combinational functions.

It is natural that backward traversal verification techniques, including the LIF-based method, perform better for sequential systems with compact reverse DFAs. Another significant difference between them and symbolic forward traversal techniques is their relationship with the unreachable region of a state-space. Consider the problem of checking the equivalence of two machines that are indeed equivalent. The traversal on the product graph is shown schematically in Figure 6.19 for both forward and backward traversal techniques. Since the forward traversal begins with the initial state(s)[1], denoted 'F' in the figure, and successively uses the transition functions to obtain the next set of states until a fixpoint is reached. In this manner, it only traverses the reachable region of the state space, marked 'R' in the figure, and does not deal with the unreachable region at all.

In contrast, the backward traversal starts from that region of the state-space where the outputs are unequal for the immediate inputs, denoted 'B' in the figure, and proceeds backwards through the predecessors at each successive step until a fixpoint is obtained. The outputs are equivalent in the complement of the region traversed. Note that while the traversed region should never intersect the reachable part of the state-space (since the two machines are assumed to be equivalent), the complexity of its computation is affected by the unreachable parts of the state-space (outside of the region marked 'R' in the figure). Since output equivalence is desired only in the reachable region for verification purposes, the backward traversal techniques are typically effective for systems with small (or nil) unreachable parts, e.g. counters. The same factors may also explain Filkorn's observation that his method can handle certain classes of synchronous circuits better than methods based on forward traversal.

Table 6.6 shows practical results for the LIF-based methods on counter circuits of various sizes, which are known to be difficult to handle with forward symbolic traversal techniques. As can be seen from the second and third columns, the LIF method for checking equivalence is competitive with Filkorn's method (taking into account the relative CPU speeds). Note also from the fourth column that this is a relatively inexpensive operation in comparison to obtaining canonical representations (which could be accomplished only up till a size of 12 due to memory limitations).

[1]Though the LIF-based algorithm is used for deterministic machines only, these comments are also valid for general methods that can handle nondeterministic machines by representing the transition relation as a Boolean formula.

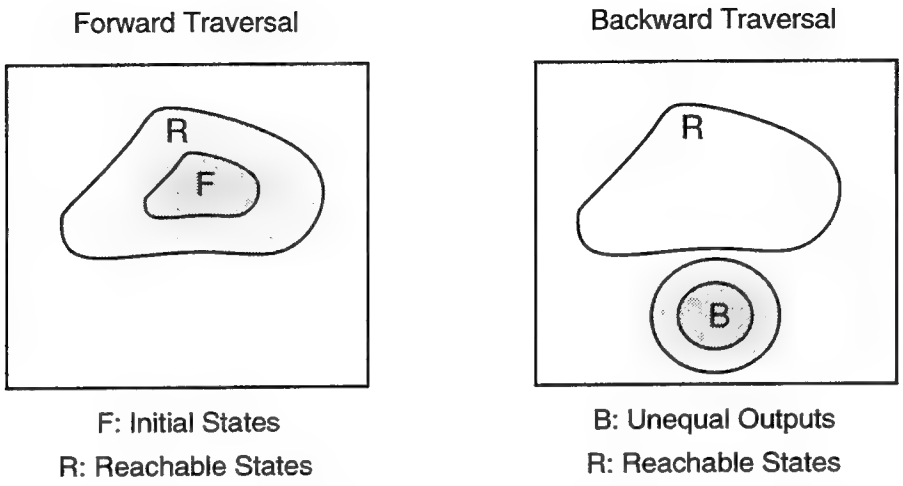
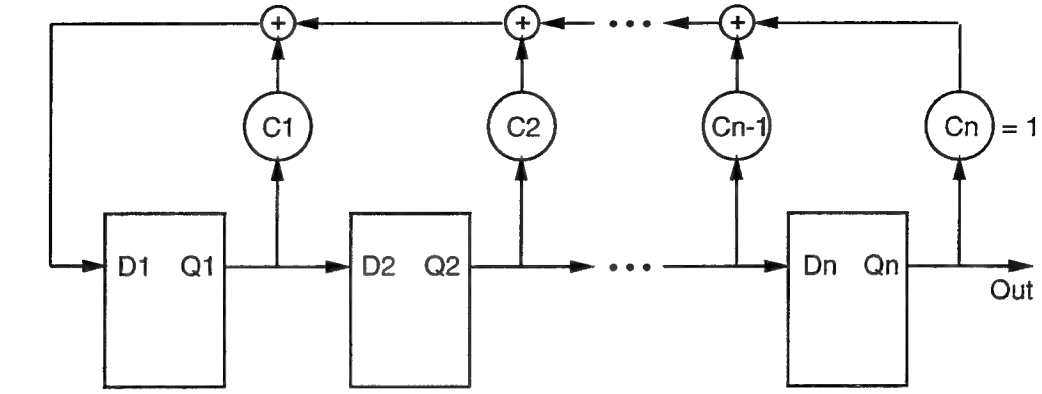


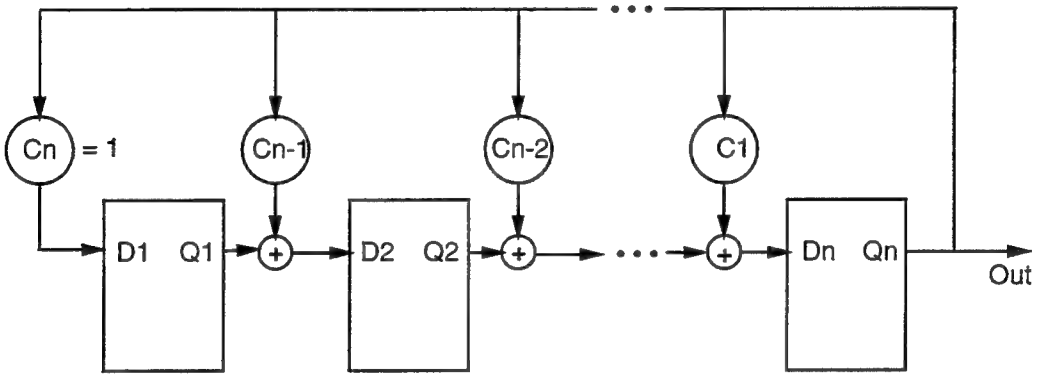
Figure 6.19: Forward and Backward Symbolic State-space Traversals

Circuit	Filkorn's method (Sun 3/60) Equivalence Time(sec.)	LIF Manipulations (SGI Indigo)	
		Equivalence Time (sec.)	Canonicity Time (sec.)
Counter 2	NA	0.3	0.3
Counter 4	NA	0.3	0.3
Counter 8	NA	0.4	2.7
Counter 16	21	0.8	—
Counter 32	66	4.1	—
Counter 64	300	31.7	—

Table 6.6: Comparison of Results for Counter Circuits



TYPE 1 LFSR



TYPE 2 LFSR

Figure 6.20: Circuits for Linear Feedback Shift Registers

In order to explore a circuit with complexity intermediate between a counter and a shift register, another set of experiments was run for linear feedback shift registers (LFSRs) of various sizes. LFSRs have numerous applications in practical systems, which include providing a source for pseudo-random binary test sequences, and carrying out response compression (used in signature analysis) [1]. Two main kinds of LFSR circuits were used in the experiments, known as Type 1 and Type 2, as shown in Figure 6.20. It is known that for the same coefficients, the two circuits produce the same periodic sequence at the output [1]. (However, for the same initial state, for an n-bit LFSR, the output of Type 2 is delayed by n with respect to the output of Type 1.) The equivalence of these two circuits was checked, where the worst-case scenario was exercised by choosing LFSR coefficients appropriately to generate maximum-length periodic sequences (length $2^n - 1$). These results are summarized in Table 6.7, where the equivalence results are shown in Column 2 and the canonicity results in Column 3.

Circuit Size	LIF Manipulations	
	Equivalence Time (sec.)	Canonicity Time (sec.)
LFSR 2	0.2	0.3
LFSR 3	0.2	0.3
LFSR 4	0.2	0.3
LFSR 5	0.2	0.4
LFSR 6	0.3	0.9
LFSR 7	0.4	2.9
LFSR 8	0.4	10.7
LFSR 9	0.6	42.1
LFSR 10	1.0	169.1
LFSR 11	1.7	—
LFSR 12	3.4	—
LFSR 13	6.7	—
LFSR 14	15.4	—
LFSR 15	41.5	—
LFSR 16	137.1	—

Table 6.7: Results for LFSR Circuits

Note that since the maximum-length output sequence requires a sequence of exponential number of states, the results for canonicity are similar to those for counter circuits. (For the same reason, forward traversal techniques are not likely to perform well for these LFSR circuits.) On the other hand, note that the results for equivalence are not as good as those for counter circuits. The primary reason is that in terms of the circuit structure, an LFSR is not as regular as a counter, thereby not allowing much of an advantage with symbolic manipulation.

6.3 Simultaneous Induction in Space and Time

The unified LIF framework for handling space and time has already been described in detail in Chapter 4, Section 4.3. The basic idea for verification proofs that combine induction in space and time is to utilize the multi-parameter canonical LIF representation as far as possible. For example, this representation was used to capture the sequential behavior of an arbitrary-length shift register circuit in Example 13, which can be used for checking its equivalence against a specification (also expressed in terms of the same parameters).

However, all cases of simultaneous parameterization in space and time cannot be handled by the finite multi-parameter LIF representation. This is not really surprising, since the intuition here is

similar to the known undecidability results for other classes of finite state representations [3, 6], (though a strict proof has not been done). The least that can be done in these cases is to parameterize in the space dimension by using LIFs, and then use symbolic LIF manipulation techniques to explicitly cover the time dimension. This technique is similar in flavor to that used for symbolic simulation of fixed-sized circuits [8, 15], and is described in detail in the following section.

6.3.1 Symbolic Simulation of Space-Parametric Circuits

As a simple example to illustrate this technique, consider the example shown earlier of a parametric accumulator (Example 14, Chapter 4), described in terms of multi-parameter functions $s^{(i,t)}$ and $c^{(i,t)}$ as follows:

for $t = 1, i \geq 1$, $s^{(i,1)} = x_{(i,1)}$
for $t > 1, i = 1$, $s^{(1,t)} = x_{(1,t)} \oplus s^{(1,t-1)}$
for $t > 1, i > 1$, $s^{(i,t)} = x_{(i,t)} \oplus s^{(i,t-1)} \oplus c^{(i-1,t)}$

for $t = 1, i = 1$, $c^{(1,1)} = 0$
for $t = 1, i > 1$, $c^{(i,1)} = x_{(i,1)} \wedge c^{(i-1,1)}$
for $t > 1, i = 1$, $c^{(1,t)} = x_{(1,t)} \wedge s^{(1,t-1)}$
for $t > 1, i > 1$, $c^{(i,t)} = (x_{(i,t)} \wedge s^{(i,t-1)}) \vee ((x_{(i,t)} \vee s^{(i,t-1)}) \wedge c^{(i-1,t)})$

Recall the observation that due to the presence of Boolean combinations of functions with non-uniform parameters, e.g. $s^{(i,t-1)} \wedge c^{(i-1,t)}$, these functions cannot be handled by multi-parameter LIF representations. Therefore, only the space dimension is retained in LIFs, which capture the space-parametric outputs explicitly at each time instant. This is shown schematically in Figure 6.21, where $s1^i, s2^i, s3^i \ldots$ and $c1^i, c2^i, c3^i \ldots$ denote the space-parametric outputs at time instants $t = 1, 2, 3, \ldots$, respectively. In the spirit of symbolic simulation, a new symbolic parameterized variable is introduced to represent the space-parametric input at each time instant, i.e. $X1_i, X2_i, X3_i \ldots$ denote the space-parameterized inputs at time instants $t = 1, 2, 3, \ldots$, respectively. Given that each of the accumulator cells shown in the figure behaves like a simple adder cell, the following definitions for the LIFs can be obtained:

for $i = 1$ $s1^1 = X1_1$
for $i > 1$ $s1^i = X1_i \oplus c1^{(i-1)}$
for $i = 1$ $c1^1 = 0$
for $i > 1$ $c1^i = X1_i \wedge c1^{(i-1)}$

for $i = 1$ $s2^1 = X2_1 \oplus s1^1$
for $i > 1$ $s2^i = X2_i \oplus s1^i \oplus c2^{(i-1)}$

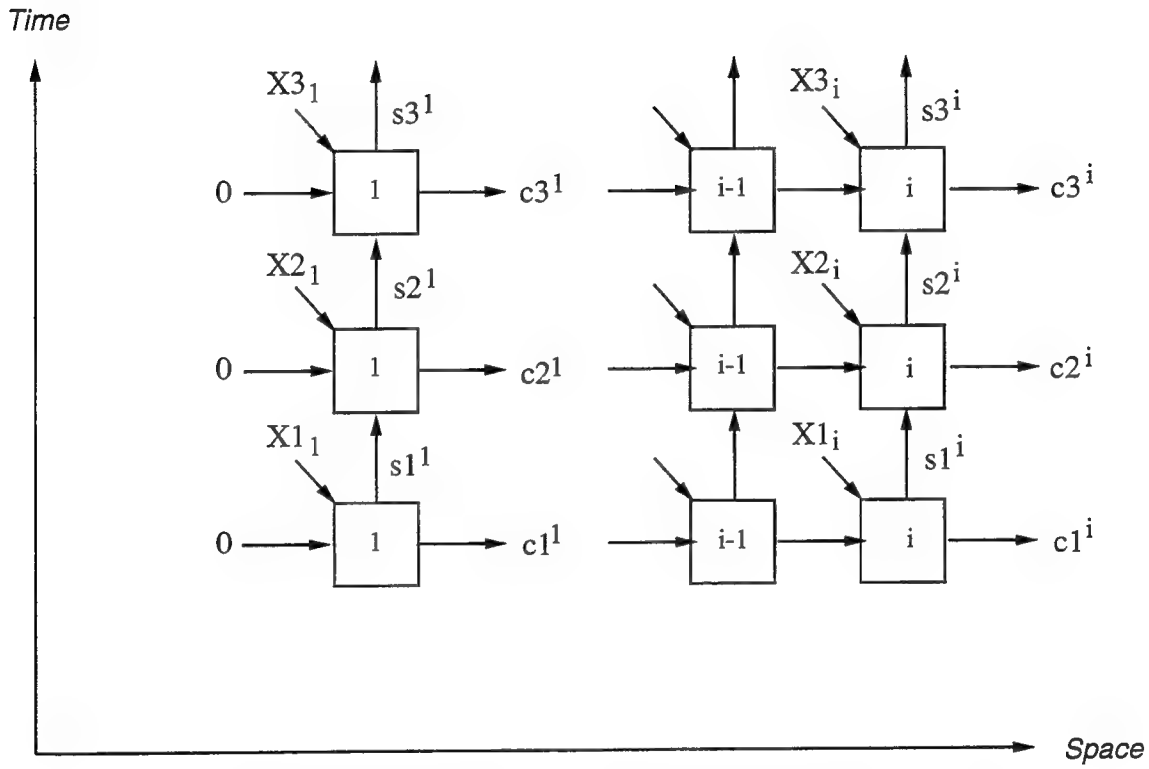


Figure 6.21: Symbolic Simulation of a Space-Parametric Accumulator

# Simulation Cycles	CPU Time (sec.)	# LIFs
2	0.5	3
4	0.5	10
6	0.6	50
8	1.0	146
10	2.3	314
12	7.1	586
14	25.3	994
16	101.7	1538
18	429.3	2218

Table 6.8: Results for Symbolic Simulation of Accumulator

for $i = 1$ $c2^1 = X2_1 \wedge s1^1$
for $i > 1$ $c2^i = (X2_i \wedge s1^i) \vee (X2_i \vee s1^i) \wedge c2^{(i-1)}$

for $i = 1$ $s3^1 = X3_1 \oplus s2^1$
for $i > 1$ $s3^i = X3_i \oplus s2^i \oplus c3^{(i-1)}$
for $i = 1$ $c3^1 = X3_1 \wedge s2^1$
for $i > 1$ $c3^i = (X3_i \wedge s2^i) \vee (X3_i \vee s2^i) \wedge c3^{(i-1)}$

Note that in the above definitions, an i-instance LIF is defined in terms of another i-instance LIF. However, handling these definitions does not pose a problem. For example, $s2^i$ is defined in terms of $s1^i$. Since $s1^i$ depends upon the i-instance variable $X1_i$ and $(i-1)$-instance LIFs, in turn, $s2^i$ also depends upon $X2_i$, $X1_i$ (both i-instance variables) and $(i-1)$-instance LIFs. This is completely consistent with an LIF definition.

By iterating this process further, LIF representations can be obtained which capture the sequential behavior of an arbitrary-sized accumulator for a fixed length of symbolic simulation. The practical results for different lengths of the simulation are shown in Table 6.8.

A practical application of this technique can be illustrated with the moving average filter example shown earlier in this chapter (Example 24, Figure 6.8). Recall that the earlier verification task was to show the equivalence of the implementation and the specification for a *fixed size* of the circuit, where the parameterization in time was utilized to check the equivalence using LIFs. Ideally, one would like to be able to utilize the simultaneous induction in space and time to verify the behavioral equivalence for all sizes of the circuit. However, simultaneous induction in space and time cannot be handled by the IBF methodology in this example (for the same reasons as described for the accumulator circuit). Therefore, consider the dual problem with a space-parametric version of the circuit (parametric in the bit-width of the adder cells, w). For a fixed depth of pipelining p, the equivalence of an arbitrary bit-width circuit can now

Pipeline Stages, p	Bit-width w	Backward	Hu & Dill	LIF Equivalence	
				w-width	all bit-widths
		Time (sec.)	Time (sec.)	Time (sec.)	Time (sec.)
2	4	NA	NA	0.6	0.3
2	8	6	5	0.7	0.3
2	16	NA	NA	0.9	0.3
3	4	NA	NA	3.0	0.7
3	8	$>$ 30 min.	47	23.2	0.7
3	16	NA	NA	153.5	0.7
4	4	NA	NA	$>$ 30 min.	64.8
4	8	$>$ 30 min.	405	$>$ 30 min.	64.8

Table 6.9: Results for Verification of All Bit-widths of Moving Average Filter

be checked against the specification by performing a symbolic simulation of length p. This is a considerable advantage over the ability to check it only for fixed bit-widths.

The practical results obtained for this task are shown in the last column of Table 6.9, where part of the Table 6.3 is shown again for comparison. Note that for each value of p, this technique performs better than the previous techniques even in the case of a 4-bit circuit. Thus, not only does it allow verification for *all* bit-widths, it also performs better than verification of particular bit-widths. This illustrates the general advantage of an induction technique, where it may be more advantageous to verify a parametric circuit by induction for all sizes, rather than handle particular circuit instances. Furthermore, by combining the parameterization in space with Hu and Dill's technique, it may be possible to verify deeper pipelines than have been reported so far.

6.3.2 Related Work

Multi-dimensional iterative networks representing simultaneous parameterization in space and time have been the target of several automatic verification efforts [6, 130, 131]. These efforts have adopted a high-level view of the system, where the verification task is formulated in terms of searching for a behavioral invariant for all sizes of the iterative network. Where possible, this invariant can be automatically obtained by checking a language containment relationship between the output languages of cell automata (for different sizes of the network). *In effect, induction in the space dimension is captured by using the language containment relationship, where the languages capture induction in the time dimension by using cell automata.* The

advantage offered by these techniques over other related work [43, 96, 155] is that the search for invariants is completely automated.

The LIF method of symbolic simulation is similar in nature. Consider first the related problem where space and time are transposed [74, 130], i.e. the space dimension is captured by cell automata, and induction along time is captured by a language containment relationship. This corresponds directly to symbolic simulation of space-parametric LIFs, where LIFs capture the (reverse) cell automata in the space dimension. By symbolic simulation of LIFs along time, the language containment relationship can be checked by using symbolic manipulations on the resulting LIF representations.

More precisely, recall that the output language produced by an FSM F can be captured as an LIF $\Gamma(F)$, as described in detail in Section 5.1.5, Chapter 5. Now, let LIFs $S_1, S_2, S_3 \ldots$ denote the outputs of the space-parametric circuit at time instants $t = 1, 2, 3, \ldots$, respectively. The corresponding languages *produced* by the cells (in the transposed network) can be obtained as $\Gamma(S_1), \Gamma(S_2), \Gamma(S_3), \ldots$, respectively. Since these languages are regular, the language containment relationship between the output languages at time instants $t = 2$ and $t = 3$ (for example), naturally translates to checking the following LIF formula:

$$\forall i.(\Gamma^i(S_3) \rightarrow \Gamma^i(S_2))$$

Note that this corresponds to tautology-checking of the LIF implication formula $(\Gamma(S_3) \rightarrow \Gamma(S_2))$. Thus, finding an induction invariant is facilitated by the automatic support for checking language containment, using tautology-checking of the LIF implication formula.

In particular, consider the language $\mathcal{L}$ defined as follows:

$$\mathcal{L} = \bigcup_{n=1}^{\omega} \Gamma(S_n)$$

From the results of previous work [6, 131] – if the language $\mathcal{L}$ is regular, which can be ascertained using LIF manipulations, then it immediately provides an invariant for the system. Furthermore, in some cases an invariant can also be found by checking for language containment with a finite union of the above languages, which can also be cast as an LIF manipulation problem.

The standard problem of finding network invariants (without space-time transpose) is handled in a similar manner, where the LIFs are defined to be parametric in time, and the symbolic simulation is carried out along the space dimension. (In the present framework of representing state transition functions as LIFs, this method can handle deterministic automata only. Some extensions using "choice" variables, as briefly mentioned in Chapter 5, require further study.)

The similarity in underlying concepts notwithstanding, there are several details that distinguish the LIF method from previous work. First, it deals uniformly with space and time, without having to explicitly transpose space and time for the cell automata, unlike previous work [130]. Second, the LIF representations correspond to *reverse* cell automata, which are more compact than the classic DFAs for several practically useful circuits. For these circuits, the LIF manipulations are potentially more efficient than classic automata techniques. For example, the

technique for obtaining the output language of an FSM (described in Section 5.1.5, Chapter 5) is different from the standard one, and does not rely on the explicit representation of the forward FSM. Third, the LIF method is fully unified within a general LIF symbolic manipulation framework. This affords a considerable advantage in providing a common substrate for trying different techniques in performing proofs by induction. Apart from spatial and temporal induction proofs, within the context of simultaneous induction in space and time, it uniformly handles different techniques such as checking equivalence of multi-parameter LIF representations (the sequence $S_1, S_2, S_3 \ldots$ converges in symbolic simulation) as well as language containment techniques (the sequence $\Gamma(S_1), \Gamma(S_2), \Gamma(S_3) \ldots$ converges in symbolic simulation).

In some sense, the advantages offered by a unified symbolic LIF manipulation framework are similar to those offered by BDD manipulation, with the extra feature of parameterization in Boolean functions that allows automatic proofs by induction where possible.

Chapter 7

Conclusions

In this thesis, a formal verification methodology has been presented for checking correctness of parametric hardware designs using automatic proofs by induction. The core of the methodology lies in symbolic manipulation of inductive Boolean functions (IBFs). These functions can be used to represent parametric hardware, as well as their specifications. A proof by induction is automatically carried out by manipulation of their symbolic representations, which are based on Binary Decision Diagrams (BDDs).

7.1 Contributions of the Thesis

The contributions of the thesis can be summarized under the following main categories:

- Symbolic Representation of Parametric Hardware
  - A characterization of two practically useful classes of inductive Boolean functions is provided, along with a representation schema for each, based on Binary Decision Diagrams. The important feature is that the complexity of the canonical representations and symbolic manipulation algorithms is independent of the induction parameter. Furthermore, the support for general symbolic manipulation translates to support for handling compositions of inductively-defined units. In combination with other useful circuit representation mechanisms, the IBF schemata are useful for parametric hardware representation in standard design libraries, with potential applications in areas other than formal verification.
  - In particular, by using time as the induction parameter, the LIF representation directly provides a canonical representation for a sequential function in terms of input sequences, in much the same way as a BDD provides a canonical representation for

a combinational function in terms of inputs. This representation, and the associated symbolic manipulation algorithms, are potentially useful in other areas dealing with sequential logic, such as synthesis, optimization, simulation and testing.

- The interesting feature of the LIF representation for a sequential function is that it corresponds to a minimal reverse DFA representation of the function. This representation is exponentially more compact than the classic forward DFA for several practical datapath circuits. Since datapath circuits are known to cause a state explosion problem in practice, the exponential compactness property is especially useful. Furthermore, the method for obtaining LIF representations (reverse DFAs) is an implicit method, which does not require construction of an explicit forward DFA. In this sense, it can be regarded as an alternative attack on the complexity of state-space representations, which form an integral part of several verification techniques.
- Though the LIF representation conceptually corresponds to a reverse DFA, it offers significant advantages with respect to a traditional state-transition graph representation of a DFA. Apart from the fact that it is a symbolic representation, it allows simultaneous exploitation of decomposition and state-sharing, unlike a traditional state-transition graph where only one or the other can be exploited. In the context of general symbolic manipulation, this translates to substantial memory savings in practice.

- Automatic Proofs by Induction for Parametric Hardware

  - With respect to formal verification, the IBF methodology provides a general framework for automating reasoning by induction. The key idea is that by building the inductive reasoning into symbolic IBF representations, a proof by induction is automatically performed by symbolic manipulation of these representations. This provides complete automation, and avoids the heuristic search typically associated with more powerful proof-theoretic frameworks.
  - The verification tasks with IBF methodology can consist of either checking equivalence of inductive representations (facilitated by their canonicity property), or checking a design correctness property (by automatic tautology-checking of the symbolic formula). This has applications in functional verification of size-parametric combinational circuits, which has traditionally been performed by theorem-provers. At the same time, it allows behavioral verification of finite state sequential systems, which has traditionally been performed by model checking and language containment techniques. Thus, the IBF methodology provides a unified framework for both structural and temporal domains.
  - Furthermore, the unified IBF methodology can be used to handle simultaneous induction in the structural and temporal domains. Applications range from checking

equivalence of multi-parameter IBF representations, with the possible addition of dummy parameters, to symbolic simulation of the temporal behavior of space-parametric circuits. The latter also allows incorporation of automatic techniques for obtaining network invariants, by checking language containment using IBF manipulations.

- Finally, a working prototype for symbolic LIF manipulation and the outlined verification methodology has been implemented. Results examining its usefulness for practical applications have also been provided. The prototype is potentially useful as a test-bed for further experimentation with existing techniques, as well as for exploring new ideas.

- General Symbolic Techniques

  While the work described in this thesis was primarily pursued towards the stated goal of automatic inductive reasoning, some associated sub-problems gave rise to very interesting sub-techniques, summarized as follows:

  - Construction of the minimal reverse DFA

    In contrast to classic DFA reversal techniques, the LIF technique for obtaining a minimal reverse DFA starts from state transition equations rather than an explicit forward DFA. The technique used is to lazily unroll the state transition equations, which directly provides the reversal. The lazy property ensures that no new states are explored unless required. The unrolling property to directly obtain reversal avoids the subset construction (required in classic techniques to determinize the NFA resulting from explicit reversal of the forward DFA). The enabling feature of this technique is the maintenance of canonicity with respect to Boolean combinations of state variables. It permits a conservative estimate of the final equivalence classes, *while* ensuring that the unrolling process actually terminates. Keeping a handle on such combinations also simplifies subsequent symbolic manipulation. In principle, maintaining such handles can be incorporated easily within classic forward DFA techniques also.

  - Obtaining the reverse DFA for the output language of an FSM

    As described in detail in Chapter 5 (Section 5.1.5), the technique for obtaining the LIF (reverse DFA) representation for the output language of an FSM has many interesting features. It utilizes the by-now standard technique [48] of introducing new variables to capture the range of a Boolean function vector. The new twist lies in pushing this operation in reverse, while taking advantage of the parametric form of the LIBDD representation. In general, the output language representation is useful in diverse applications such as handling incompletely specified machines, checking language containment among networks of processes etc.

- Parameterization techniques

  The geometric hyperspace framework (described in Chapter 4) is also a very general technique for a region-wise representation of a multi-parameter function. BDD-style mode ordering and reduction principles were utilized to provide canonicity of a simple parameter decision tree representation, with nodes that represent partitions in terms of hyperplanes. In fact, the node ordering requirement is an important factor which affects the expressiveness, and thereby the potential efficiency of this framework. For example, one may allow only basis-value partitions (for simple induction trajectories), or only non-intersecting partitions (for typical multiple output functions), or arbitrary partitions in the general case.

  Another interesting technique was the use of binary-encoding for representation of an output index. This turned out to be useful both for representation of an exponential size tree circuit in a linear form (Section 4.4.2, Chapter 4), and for changing the domain of quantification from integers to Boolean-valued variables in a specification property (Section 6.1.2, Chapter 6).

  Finally, the introduction of dummy parameters (described in Section 4.3.2. Chapter 4) is also very useful. It has been used in various flavors, such as in fanin-splitting for more efficient representations [33]. In the context of IBFs, it allows the unidirectional framework to be naturally extended to bidirectional circuits with two fixed boundaries, such as a parametric shift register.

7.2 Higher-level Perspective: Strengths and Limitations

In terms of the broad picture, the adopted approach of combining inductive reasoning with BDD-based symbolic representations has several advantages. First, the inductive reasoning is completely automatic, thereby freeing the user from the burden of setting up useful lemmas and induction hypotheses. Second, by placing the inductive reasoning at the level of symbolic representations, a general framework is provided for conducting various kinds of induction proofs in a unified manner. Traditionally, structural induction proofs have been performed by theorem-proving techniques, while temporal induction proofs have been performed by model checking/language containment techniques. The IBF methodology provides a unified framework for both. Third, admittedly some LIF manipulations correspond to techniques that have been studied already (such as backward symbolic traversal for checking FSM input/output equivalence, using language containment for obtaining network invariants etc.). However, the important point is that the inductive reasoning has been moved away from the *meta-theoretic level* to the proof-theoretic level, by incorporating it operationally with symbolic manipulation. Therefore, a new meta-theoretic justification is not required for each different technique. Rather, the same general-purpose induction facility with IBF representations is used to incorporate many

different flavors of induction proofs. Thus, symbolic IBF manipulations offer the advantages of a common substrate similar to BDD manipulations, with the extra feature of inductive parameterization which supports automatic proofs by induction.

Another useful direction that stemmed out of the thesis research is its application in the area of symbolic sequential function manipulation. The correspondence between BDDs (capturing canonicity over inputs) and LIF representations (capturing canonicity over regular input sequences) instantly provides a framework for extending BDD manipulation techniques to sequential functions. Though such techniques have already been employed with considerable success, they are based on iteration over combinational Boolean functions, in order to capture the iterative temporal behavior of sequential systems. Furthermore, the necessity of using iterative combinational analysis enforces a fixed state-encoding of a sequential system. In contrast, the LIF schema directly accords a first-class status to sequential functions, where the canonicity is captured in terms of the underlying sequences of inputs. In particular, the canonicity property is independent of the number of states as well as the state-encoding. In this light, the issue of BDD vs. LIF manipulation for sequential functions may seem like a space-time tradeoff. However, in principle, the LIF schema provides a useful implementation of the well-known functional semantics of the theory of sequential functions. In practice, it can naturally benefit from any techniques, both existing as well as new, which alleviate the space requirements for symbolic BDD manipulation in general.

The correspondence of the LIF representation of a sequential function with a reverse DFA also turned out to be very interesting. Though several efforts have been aimed in the past at finding compact state-space representations, the potential of reverse automata went virtually unnoticed. At best, they were used to obtain linear-sized BDD representations for iterative array outputs [130]. In particular, their exponential compactness advantage for practical datapath circuits had not been explored at all. Furthermore, as mentioned earlier, the LIF method of obtaining reverse DFAs is a new technique, which involves lazily unrolling state transition equations until all required states are available.

The practical results obtained so far demonstrate that the IBF methodology is effective for parametric combinational circuits, and competitive with many state-of-the-art techniques for finite state sequential systems. However, more work is required to fine-tune the prototype implementation before a realistic attempt can be made to handle industry-strength examples. As is well-known, effective BDD heuristics and optimizations can frequently spell the difference between success and failure in such cases.

Another unaddressed practical issue regards the availability of inductive descriptions of hardware. While this is not a problem with fixed-size sequential system descriptions, it is a more serious issue with size-parametric hardware. At the higher levels, many hardware description languages (HDLs) provide some sort of loop constructs for describing iterative structure [118, 140]. However, some additional support is needed to identify those subsets which follow the restrictions imposed by the IBFs. This could perhaps be facilitated by having ex-

plicit constructs for the basis and the inductive definitions (in the style of "init" and "next" of SMV [109]). At the lower levels of hardware description, it should be possible, in principle, to extract inductive descriptions from well-structured layouts, gate netlists etc. Some preliminary work in this direction has been reported [120]. However, this problem remains mostly unexplored.

Another larger concern stems from practical applications of formal verification techniques in general. Given that no single technique can handle all aspects of a complete hardware design available today, a necessary requirement is the use of appropriate abstractions. Towards that end, it is helpful both to incorporate the verification process within the design cycle itself, and to utilize the "design knowledge" wherever possible. In fact, all successful applications of formal verification in industrial contexts have relied on such information, manually obtained in most cases. It would be very useful to provide methodological mechanisms for the systematic acquisition and exploitation of such information that designers typically use. Currently, the IBF methodology does not provide any support in this direction. On the other hand, there is potential for its incorporation within the design cycle, e.g. by integrating its use within an HDL-based design development. This is likely to be more effective than post-design verification.

7.3 Directions for Future Research

The full potential of the unified LIF schema for simultaneous induction in the structural and temporal domains has not been explored yet. There are several useful directions that can be taken in this context. First, the correspondence between LIF symbolic manipulations and language containment techniques (described in Chapter 6) provides a useful starting point for further investigation into this area. The payoff in terms of automatic generation of network invariants is very useful in practice.

Second, a potential application area for simultaneous induction is in digital signal processing, where acyclic LIF representations can be used to capture the behavior of finite input response filters. Perhaps this acyclic property can be exploited to simplify the interactions of the structural and temporal induction parameters, in order to verify the circuits for all sizes of the bit-widths.

Third, model checking techniques have also not been explored yet. Given the fact that classic temporal operators require a backward traversal on the state-space, similar to that used by the LIF algorithms, it would be interesting to use the LIF schema to perform model checking on structurally-inductive sequential systems. However, since LIFs capture all values of induction parameters, it is not clear if and when the fixpoint computations will converge. Perhaps a subset of "regular" structures can be identified that will guarantee such convergence.

A different direction for future research is in introducing inductive parameterization frameworks within other canonical representations. Some good candidates are the Binary Moment

Diagrams recently proposed by Bryant and Chen [29], and polynomial representations using Zero-Suppressed BDDs proposed by Minato [114, 115]. Given the advantages of representing arithmetic (rather than Boolean only) expressions provided by these representations, it may allow automation of induction at higher levels of abstraction. As a matter of fact, most BDD enhancements studied by researchers are orthogonal to the described inductive framework, potentially allowing an easy integration with whatever is most effective for a particular application.

Within the context of IBFs, yet another direction is in establishing the equivalence of IBF representations across the different induction schema. At present, canonicity for the representation of an IBF is ensured only within its own class of functions. Given these representations, it does not seem very difficult to augment them with the capability to symbolically manipulate integer parameter expressions, in order to prove their equivalence in an automatic (though deliberative) manner. Such manipulation can be accomplished quite easily by existing theorem-provers, and the approach should be to adopt them as far as possible within the framework of completely automatic IBF manipulations. There also exists scope for defining new classes of IBFs, as and when the need might arise. The similarities between the schemata for LIFs and EIFs explored in this thesis can potentially serve as general principles for some new classes also.

Bibliography

[1] ABRAMOVICI, M., BREUER, M. A., AND FRIEDMAN, A. D. *Digital Systems Testing and Testable Design*. Computer Science Press, New York, 1990.

[2] AHO, A. V., HOPCROFT, J. E., AND ULLMAN, J. D. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Publishing Company, Reading, MA, 1974.

[3] APT, K., AND KOZEN, D. Limits for automatic verification of finite-state concurrent systems. *Information Processing Letters 22*, 6 (1986), 307–309.

[4] ASHAR, P., AND CHEONG, M. Efficient breadth-first manipulation of Binary Decision Diagrams. In *Proceedings of the IEEE International Conference on Computer-Aided Design* (1994), IEEE Computer Society Press, Los Alamitos, CA, pp. 622–627.

[5] AZIZ, A., BALARIN, F., CHENG, S.-T., HOJATI, R., KAM, T., KRISHNAN, S. C., RANJAN, R. K., SHIPLE, T., SINGHAL, V., TASIRAN, S., WANG, H.-Y., BRAYTON, R. K., AND SANGIVANNI-VINCENTELLI, A. HSIS: A BDD-based environment for formal verification. In *Proceedings of the 31st ACM/IEEE Design Automation Conference* (June 1994), IEEE Computer Society Press, Los Alamitos, CA, pp. 454–459.

[6] BALARIN, F., AND SANGIOVANNI-VINCENTELLI, A. On the automatic computation of network invariants. In *Proceedings of the Conference on Computer-Aided Verification* (June 1994), vol. 818 of *Lecture Notes in Computer Science*, Springer-Verlag, New York, pp. 234–246.

[7] BEATTY, D. L. *A Methodology for Formal Hardware Verification, with Application to Microprocessors*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, Aug. 1993.

[8] BEATTY, D. L., BRYANT, R. E., AND SEGER, C. H. Synchronous circuit verification by symbolic simulation: An illustration. In *Proceedings of the Sixth MIT Conference on Advanced Research in VLSI* (1990), W. J. Dally, Ed., MIT Press, Cambridge, pp. 98–112.

[9] BERGSTRA, J., HEERING, J., AND KLINT, P. *Algebraic Specification.* Addison-Wesley, New York, NY, 1989.

[10] BERMAN, C. L. Circuit width, register allocation, and reduced function graphs. Tech. Rep. RC 14129, IBM T. J. Watson Research Center, Yorktown Heights, NY, 1988.

[11] BERMAN, C. L. Ordered Binary Decision Diagrams and circuit structure. In *Proceedings of the IEEE International Conference on Computer Design* (1989), IEEE Computer Society Press, Silver Spring, MD, pp. 392–395.

[12] BILLON, J. P. Perfect normal forms for discrete functions. Tech. Rep. 87019, Bull Research Center, Louveciennes, France, June 1987.

[13] BLUM, M., CHANDRA, A. K., AND WEGMAN, M. N. Equivalence of free boolean graphs can be decided probabilistically in polynomial time. *Information Processing Letters 10(2)* (Mar. 1980), 80–82.

[14] BOLLIG, B., SAUERHOFF, M., SIELING, D., AND WEGENER, I. Read k times ordered binary decision diagrams – efficient algorithms in the presence of null chains. Tech. Rep. 47, Universitat Dortmund, 44221 Dortmund, Germany, 1993.

[15] BOSE, S., AND FISHER, A. L. Verifying pipelined hardware using symbolic logic simulation. In *Proceedings of the IEEE International Conference on Computer Design* (1989), IEEE Computer Society Press, Silver Spring, MD, pp. 217–221.

[16] BOSE, S., AND FISHER, A. L. Automatic verification of synchronous circuits using symbolic simulation and temporal logic. In *Proceedings of the IFIP International Workshop on Applied Formal Methods for Correct VLSI Design, Belgium, 1989* (1990), L. M. J. Claesen, Ed., vol. II, North-Holland, Amsterdam, pp. 151–158.

[17] BOYER, R. S., AND MOORE, J. S. *A Computational Logic Handbook.* Academic Press, Boston, 1988.

[18] BRACE, K. S., RUDELL, R. L., AND BRYANT, R. E. Efficient implementation of a BDD package. In *Proceedings of the 27th ACM/IEEE Design Automation Conference* (June 1990), IEEE Computer Society Press, Los Alamitos, CA, pp. 40–45.

[19] BRAUER, W. On minimizing finite automata. *Bulletin of the European Association for Theoretical Computer Science 35* (June 1988), 113–116.

[20] BRAYTON, R. K., HACHTEL, G. D., MCMULLEN, C., AND SANGIOVANNI-VINCENTELLI, A. *Logic Minimization Algorithms for VLSI Synthesis.* Kluwer Academic Publishers, Boston, 1984.

[21] BRENT, R. P., AND KUNG, H. T. A regular layout for parallel adders. *IEEE Transactions on Computers C-31* (Mar. 1982), 260–264.

[22] BRYANT, R. E. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers C-35*, 8 (Aug. 1986), 677–691.

[23] BRYANT, R. E. Algorithmic aspects of symbolic switch network analysis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 6*, 4 (July 1987), 618–633.

[24] BRYANT, R. E. A methodology for hardware verification based on logic simulation. Tech. Rep. CMU-CS-87-128, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, June 1987.

[25] BRYANT, R. E. Symbolic analysis of VLSI circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 6*, 4 (July 1987), 634–649.

[26] BRYANT, R. E. On the complexity of VLSI implementations and graph representations of Boolean functions with application to integer multiplication. *IEEE Transactions on Computers 40*, 2 (Feb. 1991), 205–213.

[27] BRYANT, R. E. Symbolic Boolean manipulation with Ordered Binary Decision Diagrams. *ACM Computing Surveys 24*, 3 (Sept. 1992), 293–318.

[28] BRYANT, R. E., BEATTY, D., BRACE, K., CHO, K., AND SHEFFLER, T. COSMOS: A compiled simulator for MOS circuits. In *Proceedings of the 24th ACM/IEEE Design Automation Conference* (June 1987), IEEE Computer Society Press, Los Alamitos, CA, pp. 9–16.

[29] BRYANT, R. E., AND CHEN, Y.-A. Verification of arithmetic functions with binary moment diagrams. Tech. Rep. CMU-CS-94-160, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, May 1994.

[30] BRYANT, R. E., AND SEGER, C. H. Formal verification of digital circuits using symbolic ternary system models. In *Proceedings of the Workshop on Computer-Aided Verification (CAV 90)* (1991), E. M. Clarke and R. P. Kurshan, Eds., vol. 3 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, American Mathematical Society, Springer-Verlag, New York.

[31] BURCH, J., CLARKE, E. M., MCMILLAN, K., DILL, D., AND HWANG, J. Symbolic model checking: 10^{20} states and beyond. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science* (June 1990), IEEE Computer Society Press, Washington, D.C., pp. 428–439.

[32] BURCH, J., CLARKE, E. M., MCMILLAN, K., AND DILL, D. L. Sequential circuit verification using symbolic model checking. In *Proceedings of the 27th ACM/IEEE Design Automation Conference* (June 1990), IEEE Computer Society Press, Los Alamitos, CA, pp. 46–51.

[33] BURCH, J. R. Using BDDs to verify multipliers. In *Proceedings of the 28th ACM/IEEE Design Automation Conference* (June 1991), IEEE Computer Society Press, Los Alamitos, CA, pp. 408–412.

[34] BURCH, J. R., CLARKE, E. M., AND LONG, D. E. Representing circuits more efficiently in symbolic model checking. In *Proceedings of the 28th ACM/IEEE Design Automation Conference* (June 1991), IEEE Computer Society Press, Los Alamitos, CA, pp. 403–407.

[35] BURCH, J. R., CLARKE, E. M., AND LONG, D. E. Symbolic model checking with partitioned transition relations. In *Proceedings of the 1991 International Conference on VLSI* (Aug. 1991), A. Halaas and P. B. Denyer, Eds.

[36] BURCH, J. R., AND DILL, D. L. Automatic verification of pipelined microprocessor control. In *Proceedings of the Workshop on Computer-Aided Verification* (June 1994), D. L. Dill, Ed., vol. 818, Springer-Verlag, New York, pp. 68–80.

[37] CAMILLERI, A. J., GORDON, M. J. C., AND MELHAM, T. F. Hardware verification using higher-order logic. In *From HDL Descriptions to Guaranteed Correct Circuit Designs*, D. Borrione, Ed. North-Holland, Amsterdam, 1987, pp. 43–67.

[38] CLARKE, E., GRUMBERG, O., HIRAISHI, H., JHA, S., LONG, D., MCMILLAN, K., AND NESS, L. Verification of Futurebus+ cache coherence protocol. In *Proceedings of the Eleventh International Symposium on Computer Hardware Description Languages and their Applications* (1993).

[39] CLARKE, E. M. Private Communication, 1994.

[40] CLARKE, E. M., AND EMERSON, E. A. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proceedings of the Workshop on Logics of Programs* (1981), vol. 131 of *Lecture Notes in Computer Science*, Springer-Verlag, New York, pp. 52–71.

[41] CLARKE, E. M., EMERSON, E. A., AND SISTLA, A. P. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems 8*, 2 (Apr. 1986), 244–263.

[42] CLARKE, E. M., FILKORN, T., AND JHA, S. Exploiting symmetry in temporal logic model checking. In *Proceedings of the Conference on Computer-Aided Verification*

(June 1993), vol. 697 of *Lecture Notes in Computer Science*, Springer-Verlag, New York, pp. 450–462.

[43] CLARKE, E. M., AND GRUMBERG, O. Avoiding the state explosion problem in temporal logic model checking algorithms. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing* (Aug. 1987), ACM, New York, pp. 294–303.

[44] CLARKE, E. M., GRUMBERG, O., AND BROWNE, M. C. Reasoning about networks with many identical finite-state processes. In *Proceedings of the Fifth Annual ACM Symposium on Principles of Distributed Computing* (Aug. 1986), ACM, New York, pp. 240–248.

[45] CLARKE, E. M., GRUMBERG, O., AND LONG, D. E. Model checking and abstraction. In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Programming Languages* (Jan. 1992), ACM, New York.

[46] CLARKE, E. M., KIMURA, S., LONG, D. E., MICHAYLOV, S., SCHWAB, S. A., AND VIDAL, J. P. Symbolic computation algorithms on shared memory multiprocessors. In *Shared Memory Multiprocessing*, N. Suzuki, Ed. MIT Press, 1992.

[47] COUDERT, O., BERTHET, C., AND MADRE, J. C. Verification of sequential machines using Boolean functional vectors. In *Proceedings of the IFIP International Workshop on Applied Formal Methods for Correct VLSI Design, Belgium, 1989* (1990), L. M. J. Claesen, Ed., vol. II, North-Holland, Amsterdam, pp. 179–196.

[48] COUDERT, O., BERTHET, C., AND MADRE, J. C. Verification of synchronous sequential machines using symbolic execution. In *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems, Grenoble, France, June 1989* (1990), vol. 407 of *Lecture Notes in Computer Science*, Springer-Verlag, New York, pp. 365–373.

[49] COUDERT, O., MADRE, J. C., AND BERTHET, C. Verifying temporal properties of sequential machines without building their state diagrams. In *Proceedings of the Workshop on Computer-Aided Verification (CAV 90)* (1991), E. M. Clarke and R. P. Kurshan, Eds., vol. 3 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, American Mathematical Society, Springer-Verlag, New York.

[50] DE GEUS, A. J. High level design: A design vision for the 90's. In *Proceedings of the IEEE International Conference on Computer Design* (1992), IEEE Computer Society Press, Silver Spring, MD, p. 8.

[51] DEVADAS, S. Optimizing interacting finite state machines using sequential don't cares. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 10*, 12 (Dec. 1991), 1473–1484.

[52] DILL, D. L., DREXLER, A. J., HU, A. J., AND YANG, C. H. Protocol verification as a hardware design aid. In *Proceedings of the IEEE International Conference on Computer Design* (Oct. 1992), pp. 522–525.

[53] EMERSON, E. A. Temporal and modal logic. In *Handbook of Theoretical Computer Science*, J. van Leeuwen, Ed., vol. B. Elsevier Science Publishers, Amsterdam, 1990, pp. 995–1071.

[54] EMERSON, E. A., AND SISTLA, A. P. Symmetry and model checking. In *Proceedings of the Conference on Computer-Aided Verification* (June 1993), vol. 697 of *Lecture Notes in Computer Science*, Springer-Verlag, New York, pp. 463–478.

[55] FILKORN, T. A method for symbolic verification of synchronous circuits. In *Proceedings of the Tenth International Symposium on Computer Hardware Description Languages and their Applications* (Apr. 1991), D. Boorione and R. Waxman, Eds., IFIP, North-Holland, Amsterdam.

[56] FISHER, A. L., AND BRYANT, R. E. Performance of COSMOS on the IFIP workshop benchmarks. In *Proceedings of the IFIP International Workshop on Applied Formal Methods for Correct VLSI Design, Belgium, 1989* (1990), L. M. J. Claesen, Ed., vol. II, North-Holland, Amsterdam, pp. 101–105.

[57] FLORES, I. *The Logic of Computer Arithmetic.* Prentice-Hall, Englewood Cliffs, N. J., 1983.

[58] FUJITA, M., FUJISAWA, H., AND KAWATO, N. Evaluation and improvements of a Boolean comparison program based on Binary Decision Diagrams. In *Proceedings of the IEEE International Conference on Computer-Aided Design* (1988), IEEE Computer Society Press, Los Alamitos, CA, pp. 2–5.

[59] GABBAY, D., AND GUENTHNER, F., Eds. *Handbook of Philosophical Logic*, vol. 1, 2 and 3. D. Reidel, Boston, 1983.

[60] GAREY, M. R., AND JOHNSON, D. S. *Computers and Intractability: A Guide to the Theory of NP-completeness.* W. H. Freeman, San Francisco, 1979.

[61] GERGOV, J., AND MEINEL, C. Mod-2-OBDDs – a generalization of OBDDs and EXOR-Sum-of-Products. Tech. rep., Universitat Trier, D-5500 Trier, Germany, 1993.

[62] GERGOV, J., AND MEINEL, C. Efficient boolean manipulation of OBDD's can be extended to FBDD's. *IEEE Transactions on Computers 43*, 10 (Oct. 1994), 1197–1209.

[63] GERMAN, S. M., AND WANG, Y. Formal verification of parameterized hardware designs. In *Proceedings of the IEEE International Conference on Computer Design* (1985), IEEE Computer Society Press, Silver Spring, MD, pp. 549–552.

[64] GORDON, M. J. C. HOL: A proof generating system for higher-order logic. In *VLSI Specification, Verification and Synthesis*, G. Birtwistle and P. A. Subrahmanyam, Eds. Kluwer Academic Publishers, Boston, 1987, pp. 73–128.

[65] GORDON, M. J. C., MILNER, R., AND WADSWORTH, C. P. *Edinburgh LCF: A Mechanized Logic of Computation*, vol. 78 of *Lecture Notes in Computer Science*. Springer-Verlag, New York, 1979.

[66] GRUMBERG, O., AND LONG, D. E. Model checking and modular verification. In *Proceedings of CONCUR '91: Second International Conference on Concurrency Theory* (Aug. 1991), vol. 527 of *Lecture Notes in Computer Science*, Springer-Verlag, New York.

[67] GUPTA, A. Formal hardware verification methods : A survey. In *Formal Methods in System Design*, R. K. Brayton, E. M. Clarke, and P. A. Subrahmanyam, Eds., vol. 1 (Nos. 2/3). Kluwer Academic Publishers, Boston, Oct. 1992.

[68] GUPTA, A., AND FISHER, A. L. Parametric circuit representation using inductive Boolean functions. In *Proceedings of the Conference on Computer-Aided Verification* (June 1993), vol. 697 of *Lecture Notes in Computer Science*, Springer-Verlag, New York, pp. 15–28.

[69] GUPTA, A., AND FISHER, A. L. Representation and symbolic manipulation of linearly inductive Boolean functions. In *Proceedings of the IEEE International Conference on Computer-Aided Design* (Nov. 1993), IEEE Computer Society Press, Los Alamitos, CA, pp. 192–199.

[70] GUPTA, A., AND FISHER, A. L. Tradeoffs in canonical sequential function representations. In *Proceedings of the IEEE International Conference on Computer Design* (Oct. 1994), IEEE Computer Society Press, Los Alamitos, CA, pp. 111–116.

[71] HACHTEL, G., RHO, J.-K., SOMENZI, F., AND JACOBY, R. Exact and heuristic algorithms for the minimization of incompletely specified state machines. In *Proceedings of the European Design Automation Conference* (1991).

[72] HAR'EL, Z., AND KURSHAN, R. P. Software for analytical development of communication protocols. Tech. rep., AT&T Bell Laboratories, Murray Hill, NJ, Jan. 1990.

[73] HATCHER, W. S. *The Logical Foundations of Mathematics*. Pergamon Press, Oxford, England, 1982.

[74] HENNIE, F. C. *Iterative Arrays of Logical Circuits*. MIT Press, Cambridge, Mass., 1961.

[75] HOARE, C. A. R. Proof of correctness of data representations. *Acta Informatica 1* (1972), 271–281.

[76] HOJATI, R., SHIPLE, T., BRAYTON, R. K., AND KURSHAN, R. P. A unified approach to language containment and fair CTL model checking. In *Proceedings of the ACM/IEEE Design Automation Conference* (June 1993), pp. 475–481.

[77] HOJATI, R., TOUATI, H., KURSHAN, R. P., AND BRAYTON, R. K. Efficient ω-Regular language containment. In *Proceedings of the Workshop on Computer-Aided Verification (CAV 92)* (1992), pp. 371–382.

[78] HOPCROFT, J. E. An $n \log n$ algorithm for minimizing states in a finite automaton. In *Theory of Machines and Computation*, Z. Kohavi and A. Paz, Eds. Academic Press, New York, 1971, pp. 189–196.

[79] HOPCROFT, J. E., AND ULLMAN, J. D. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley Publishing Company, Reading, MA, 1979.

[80] HU, A. J., AND DILL, D. L. Efficient verification with BDDs using implicitly conjoined invariants. In *Proceedings of the Conference on Computer-Aided Verification* (June 1993), vol. 697 of *Lecture Notes in Computer Science*, Springer-Verlag, New York, pp. 3–14.

[81] HU, A. J., DILL, D. L., DREXLER, A. J., AND YANG, C. H. Higher-level specification and verification with BDDs. In *Proceedings of the Workshop on Computer-Aided Verification* (June 1992), G. V. Bochmann and D. K. Probst, Eds., Springer-Verlag, New York, pp. 82–95.

[82] HUFFMAN, D. A. The synthesis of sequential switching circuits. *J. Franklin Institute 257* (Mar. 1954), 161–190. Reprinted in E. F. Moore (Ed.) *Sequential Machines: Selected Papers*, Addison Wesley Publishing Company, 1964.

[83] HUNT, JR., W. A. FM 8501: A verified microprocessor. PhD thesis, Technical Report ICSCA-CMP-47, University of Texas at Austin, 1985.

[84] HUNT, JR., W. A. The mechanical verification of a microprocessor design. In *From HDL Descriptions to Guaranteed Correct Circuit Designs*, D. Borrione, Ed. North-Holland, Amsterdam, 1987, pp. 89–129.

[85] HUNT, JR., W. A. Microprocessor design verification. *Journal of Automated Reasoning 5*, 4 (1989), 429–460.

[86] IP, C. W., AND DILL, D. Better verification through symmetry. In *Proceedings of the International Symposium on Computer Hardware Description Languages and their Applications* (Apr. 1993), IFIP, North-Holland, Amsterdam.

[87] JAIN, J., ABADIR, M., BITNER, J., FUSSELL, D., AND ABRAHAM, J. A. IBDDs: An efficient functional representation for digital circuits. In *Proceedings of the European Design Automation Conference* (May 1992), IEEE Computer Society Press, Los Alamitos, CA, pp. 440–446.

[88] JAIN, J., BITNER, J., FUSSELL, D., AND ABRAHAM, J. Probabilistic verification of Boolean functions. *Formal Methods in System Design 1* (July 1992), 63–118.

[89] JOSKO, B. Verifying the correctness of AADL-Modules using model checking. In *Proceedings of the REX Workshop on Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*, J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, Eds., vol. 430 of *Lecture Notes in Computer Science*. Springer-Verlag, New York, 1989.

[90] JOYCE, J. J., AND SEGER, C.-J. H. Linking BDD-Based symbolic evaluation to interactive theorem-pproving. In *Proceedings of the 30th ACM/IEEE Design Automation Conference* (June 1993), IEEE Computer Society Press, Los Alamitos, CA, pp. 469–474.

[91] KEISTER, W., RITCHIE, A. E., AND WASHBURN, S. H. *The Design of Switching Circuits*. Van Nostrand, New York, 1951.

[92] KIM, J., AND NEWBORN, M. M. The simplification of sequential machines with input restrictions. *IEEE Transactions on Computers C-20*, 12 (Dec. 1972), 1440–1443.

[93] KOHAVI, Z. *Switching and Finite Automata Theory*. McGraw-Hill, New York, 1978.

[94] KRISCHER, S. The backward walk approach in FSM verification. In *Proceedings of the International Symposium on Computer Hardware Description Languages and their Applications* (Apr. 1993), IFIP, North-Holland, Amsterdam.

[95] KURSHAN, R. P. Analysis of discrete event coordination. In *Proceedings of the REX Workshop on Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*, J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, Eds., vol. 430 of *Lecture Notes in Computer Science*. Springer-Verlag, New York, 1989.

[96] KURSHAN, R. P., AND MCMILLAN, K. L. A structural induction theorem for processes. In *Proceedings of the Eighth Annual ACM Symposium on Principles of Distributed Computing* (1989), ACM, New York, pp. 239–247.

[97] KURSHAN, R. P., MERRITT, M., ORDA, A., AND SACHS, S. A structural linearization principle for processes. In *Proceedings of the Conference on Computer-Aided Verification* (June 1993), vol. 697 of *Lecture Notes in Computer Science*, Springer-Verlag, New York, pp. 491–504.

[98] LICHTENSTEIN, O., AND PNUELI, A. Checking that finite state concurrent programs satisfy their linear specifications. In *Proceedings of the Twelfth Annual ACM Symposium on Principles of Programming Languages* (1985), ACM, New York, pp. 97–107.

[99] LIN, B., AND SOMENZI, F. Minimization of symbolic relations. In *Proceedings of the IEEE International Conference on Computer-Aided Design* (1990), IEEE Computer Society Press, Los Alamitos, CA, pp. 88–91.

[100] LIN, B., TOUATI, H. J., AND NEWTON, A. R. Don't care minimization of multi-level sequential logic networks. In *Proceedings of the IEEE International Conference on Computer-Aided Design* (1990), IEEE Computer Society Press, Los Alamitos, CA, pp. 414–417.

[101] LISANKE, R., Ed. *FSM Benchmark Suite*. Microelectronics Center of North Carolina, Research Triangle Park, North Carolina, 1987.

[102] LISKOV, B., AND GUTTAG, J. *Abstraction and Specification in Program Development*. MIT Press, Cambridge, 1986.

[103] LONG, D. E. BDD Package. Unpublished, 1993. School of Computer Science, Carnegie Mellon UniversityPittsburgh, PA.

[104] LONG, D. E. *Model Checking, Abstraction and Modular Verification*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, July 1993.

[105] MALIK, S., WANG, A. BRAYTON, R. K., AND SANGIOVANNI-VINCENTELLI, A. Logic verification using Binary Decision Diagrams in a logic synthesis environment. In *Proceedings of the IEEE International Conference on Computer-Aided Design* (1988), IEEE Computer Society Press, Los Alamitos, CA, pp. 6–9.

[106] MANNA, Z., AND PNUELI, A. Verification of concurrent programs: The temporal framework. In *Correctness Problem in Computer Science*, R. S. Boyer and J. S. Moore, Eds. Academic Press, London, 1982, pp. 215–273.

[107] MCCLUSKEY, E. J. Iterative combinational switching networks: General design considerations. *IRE Transactions on Electronic Computers EC-7* (Dec. 1958), 285–291.

[108] MCMILLAN, K. L. *Symbolic Model Checking, An approach to the state explosion problem.* PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, May 1992.

[109] MCMILLAN, K. L. The SMV System. Unpublished, 1992. School of Computer Science, Carnegie Mellon UniversityPittsburgh, PA.

[110] MCMILLAN, K. L., AND SCHWALBE, J. Formal verification of the Encore Gigamax cache consistency protocol. In *Proceedings of the International Symposium on Shared Memory Multiprocessors* (1991).

[111] MEAD, C. A., AND CONWAY, L. A. *Introduction to VLSI Systems.* Addison-Wesley, Reading, MA, 1980.

[112] MELHAM, T. F. Using recursive types to reason about hardware in higher order logic. In *Fusion of Hardware Design and Verification*, G. J. Milne, Ed. North-Holland, Amsterdam, 1988, pp. 27–50.

[113] MILNER, R. *A Calculus of Communicating Systems*, vol. 92 of *Lecture Notes in Computer Science.* Springer-Verlag, New York, 1980.

[114] MINATO, S.-I. Zero-suppressed BDDs for set manipulation in combinatorial problems. In *Proceedings of the ACM/IEEE Design Automation Conference* (June 1993), pp. 272–277.

[115] MINATO, S.-I. Implicit manipulation of polynomials using Zero-suppressed BDDs. Unpublished, 1994. NTT LSI Laboratories, Japan.

[116] MINATO, S.-I., ISHIURA, N., AND YAJIMA, S. Shared Binary Decision Diagram with attributed edges for efficient boolean function manipulation. In *Proceedings of the ACM/IEEE Design Automation Conference* (June 1990), pp. 52–57.

[117] MORGAN, C. P., AND JARVIS, D. B. Transistor logic using current switching and routing techniques and its application to a fast-carry propagation adder. *Proceedings Institute of Electrical Engineers Part B, Volume 106* (Sept. 1959), 467–468.

[118] NAVABI, Z. *VHDL – Analysis and Modeling of Digital Systems.* McGraw-Hill Inc., 1993.

[119] OCHI, H., YASUOKA, K., AND YAJIMA, S. Breadth-first manipulation of very large Binary Decision Diagrams. In *Proceedings of the IEEE International Conference on Computer-Aided Design* (1993), IEEE Computer Society Press, Los Alamitos, CA, pp. 48–55.

[120] OHMURA, M., YASUURA, H., AND TAMARU, K. Extraction of functional information from combinational circuits. In *Proceedings of the IEEE International Conference on Computer-Aided Design* (1990), IEEE Computer Society Press, Los Alamitos, CA, pp. 176–179.

[121] OWRE, S., RUSHBY, J. M., AND SHANKAR, N. PVS: A prototype verification system. In *Proceedings of the 11th International Conference on Automated Deduction*, D. Kapur, Ed., vol. 607 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, New York, 1992, pp. 748–752.

[122] PAULL, M. C., AND UNGER, S. H. Minimizing the number of states in incompletely specified sequential circuits. *IRE Transactions on Electronic Computers EC-8* (Sept. 1959), 356–357.

[123] PIXLEY, C. A computational theory and implementation of sequential hardware equivalence. In *Proceedings of the Workshop on Computer-Aided Verification (CAV 90)* (1991), E. M. Clarke and R. P. Kurshan, Eds., vol. 3 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, American Mathematical Society, Springer-Verlag, New York.

[124] PIXLEY, C. A theory and implementation of sequential hardware equivalence. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 11*, 12 (Dec. 1992), 1469–1478.

[125] PNUELI, A. In transition from global to modular temporal reasoning about programs. In *Logics and Models of Concurrent Systems*, K. Apt, Ed., vol. 13 of *NATO ASI series, Series F, Computer and System Sciences*. Springer-Verlag, New York, 1984, pp. 123–144.

[126] PREPARATA, F. P., AND SHAMOS, M. I. *Computational Geometry: An Introduction*. Springer-Verlag, New York, 1985.

[127] QUEILLE, J. P., AND SIFAKIS, J. Specification and verification of concurrent systems in CESAR. In *Proceedings of the Fifth International Symposium in Programming* (1982), vol. 137 of *Lecture Notes in Computer Science*, Springer-Verlag, New York, pp. 337–351.

[128] RABIN, M., AND SCOTT, D. Finite automata and their decision problems. *IBM Journal of Research and Development 3(2)* (1959), 115–125.

[129] RHO, J.-K., HACHTEL, G., AND SOMENZI, F. Don't care sequences and the optimization of interacting finite state machines. In *Proceedings of the IEEE International Conference on Computer-Aided Design* (1991), IEEE Computer Society Press, Los Alamitos, CA, pp. 418–421.

[130] RHO, J.-K., AND SOMENZI, F. Inductive verification of iterative systems. In *Proceedings of the 29th ACM/IEEE Design Automation Conference* (June 1992), IEEE Computer Society Press, Los Alamitos, CA, pp. 628–633.

[131] RHO, J.-K., AND SOMENZI, F. Automatic generation of network invariants for iterative sequential systems. In *Proceedings of the Conference on Computer-Aided Verification* (June 1993), vol. 697 of *Lecture Notes in Computer Science*, Springer-Verlag, New York, pp. 123–137.

[132] RUDELL, R. Dynamic variable ordering for Ordered Binary Decision Diagrams. In *Proceedings of the IEEE International Conference on Computer-Aided Design* (1993), IEEE Computer Society Press, Los Alamitos, CA, pp. 42–47.

[133] SENTOVICH, E. M., SINGH, K. J., MOON, C., SAVOJ, H., BRAYTON, R. K., AND SANGIOVANNI-VINCENTELLI, A. Sequential circuit design using synthesis and optimization. In *Proceedings of the IEEE International Conference on Computer Design* (1992).

[134] SHEN, A., DEVADAS, S., AND GHOSH, A. Probabilistic construction and manipulation of free Boolean diagrams. In *Proceedings of the IEEE International Conference on Computer-Aided Design* (1993), IEEE Computer Society Press, Los Alamitos, CA, pp. 554–549.

[135] SHTADLER, Z., AND GRUMBERG, O. Network grammars, communication behaviors and automatic verification. In *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems, Grenoble, France, June 1989* (1990), vol. 407 of *Lecture Notes in Computer Science*, Springer-Verlag, New York, pp. 151–165.

[136] SISTLA, A. P., CLARKE, E. M., FRANCEZ, N., AND MEYER, A. M. Can message buffers be axiomatized in temporal logic? *Information and Control 63*, 1 (1984), 88–112.

[137] SISTLA, A. P., AND GERMAN, S. Reasoning with many processes. In *Proceedings of the Annual Symposium on Logic in Computer Science* (1987), IEEE Computer Society Press, Washington, D.C., pp. 138–152.

[138] SKALANSKY, J. Conditional-sum addition logic. *IRE Transactions on Electronic Computers EC-9* (June 1960), 691–698.

[139] SWARTZLANDER, E. E. *Computer Arithmetic*. Van Nostrand, New York, NY, 1980.

[140] THOMAS, D. E., AND MOORBY, P. *The Verilog Hardware Description Language*. Kluwer Academic Publishers, Boston, 1991.

[141] THOMAS, W. Automata on infinite objects. In *Handbook of Theoretical Computer Science*, J. van Leeuwen, Ed., vol. B. Elsevier Science Publishers, Amsterdam, 1990, pp. 133–191.

[142] TOUATI, H. J., SAVOJ, H., LIN, B., BRAYTON, R. K., AND SANGIVANNI-VINCENTELLI, A. Implicit state enumeration of finite state machines using BDDs. In *Proceedings of the IEEE International Conference on Computer-Aided Design* (1990), IEEE Computer Society Press, Los Alamitos, CA, pp. 130–133.

[143] UNGER, S. H. *Asynchronous Sequential Switching Circuits*. John Wiley, New York, 1969.

[144] UNGER, S. H. The generation of completion signals in iterative combinational circuits. *IEEE Transactions on Computers C-26*, 1 (Jan. 1977), 13–18.

[145] UNGER, S. H. Tree realizations of iterative circuits. *IEEE Transactions on Computers C-26*, 4 (Apr. 1977), 365–383.

[146] VAN DE SNEUPSCHEUT, J. L. A. *Trace Theory and VLSI Design*. PhD thesis, Department of Computing Science, Eindhoven University of Technology, The Netherlands, 1983.

[147] VARDI, M., AND WOLPER, P. Automata-theoretic techniques for modal logics of programs. *Journal of Computer and System Sciences 32*, 2 (Apr. 1986), 183–221.

[148] VERKEST, D., CLAESEN, L., AND DE MAN, H. Special benchmark session on formal system design. In *Proceedings of the IFIP International Workshop on Applied Formal Methods for Correct VLSI Design, Belgium, 1989* (1990), L. M. J. Claesen, Ed., vol. I, North-Holland, Amsterdam, pp. 413–414.

[149] VERKEST, D., JOHANNES, P., CLAESEN, L., AND DE MAN, H. Formal techniques for proving correctness of parameterized hardware using correctness preserving transformations. In *Fusion of Hardware Design and Verification*, G. J. Milne, Ed. North-Holland, Amsterdam, 1988, pp. 77–97.

[150] VERKEST, D., JOHANNES, P., CLAESEN, L., AND DE MAN, H. Correctness proofs of parameterized hardware modules in the Cathedral-II synthesis environment. In *Proceedings of the European Design Automation Conference Glasgow* (1990), IEEE Computer Society Press, Washington, D.C.

[151] WALLACE, C. S. Conditional-sum addition logic. *IEEE Transactions on Computers EC-13* (Feb. 1964), 14–17.

[152] WANG, H.-Y., AND BRAYTON, R. K. Input don't care sequences in fsm networks. In *Proceedings of the IEEE International Conference on Computer-Aided Design* (1993), IEEE Computer Society Press, Los Alamitos, CA, pp. 321–328.

[153] WIRSING, M. Algebraic specification. In *Handbook of Theoretical Computer Science*, J. van Leeuwen, Ed., vol. B. Elsevier Science Publishers, Amsterdam, 1990.

[154] WOLPER, P. Expressing interesting properties of programs in propositional temporal logic. In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Programming Languages* (Jan. 1986), ACM, New York, pp. 184–192.

[155] WOLPER, P., AND LOVINFOSSE, V. Verifying properties of large sets of processes with network invariants. In *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems, Grenoble, France, June 1989* (1990), vol. 407 of *Lecture Notes in Computer Science*, Springer-Verlag, New York, pp. 68–80.

[156] WOLPER, P., VARDI, M. Y., AND SISTLA, A. P. Reasoning about infinite computation paths. In *Proceedings of the 24th Annual Symposium on Foundations of Computer Science* (1983), IEEE, New York, pp. 185–194.